Scalable approximate inference methods for Bayesian deep learning

Julian Hippolyt Ritter

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

March 20, 2023

I, Julian Hippolyt Ritter, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

This thesis proposes multiple methods for approximate inference in deep Bayesian neural networks split across three parts.

The first part develops a scalable Laplace approximation based on a block-diagonal Kronecker factored approximation of the Hessian. This approximation accounts for parameter correlations – overcoming the overly restrictive independence assumption of diagonal methods – while avoiding the quadratic scaling in the number of parameters of the full Laplace approximation. The chapter further extends the method to online learning where datasets are observed one at a time. As the experiments demonstrate, modelling correlations between the parameters leads to improved performance over the diagonal approximation in uncertainty estimation and continual learning, in particular in the latter setting the improvements can be substantial.

The second part explores two parameter-efficient approaches for variational inference in neural networks, one based on factorised binary distributions over the weights, one extending ideas from sparse Gaussian processes to neural network weight matrices. The former encounters similar underfitting issues as mean-field Gaussian approaches, which can be alleviated by a MAP-style method in a hierarchical model. The latter, based on an extension of Matheron's rule to matrix normal distributions, achieves comparable uncertainty estimation performance to ensembles with the accuracy of a deterministic network while using only 25% of the number of parameters of a single ResNet-50.

The third part introduces TyXe, a probabilistic programming library built on top of Pyro to facilitate turning PyTorch neural networks into Bayesian ones. In contrast

to existing frameworks, TyXe avoids introducing a layer abstraction, allowing it to support arbitrary architectures. This is demonstrated in a range of applications, from image classification with torchvision ResNets over node labelling with DGL graph neural networks to incorporating uncertainty into neural radiance fields with PyTorch3d.

Impact Statement

The methods described in this thesis have already achieved academic impact and have the potential for further practical impact, as uncertainty estimation is becoming an increasingly important component of safety-critical machine learning systems. Most parts of this thesis have been published in leading machine learning conferences, with the contents of Chapter 3 in particular being widely referenced and having inspired various independent follow-up works. As these approaches are applicable to pre-trained deterministic neural networks, they allow for introducing Bayesian uncertainty estimates into already existing systems with minimal overhead. The memory-efficient techniques of Chapter 4 are a step towards bringing uncertainty estimation to resource-constrained settings such as on edge devices, when data must be processed locally, e.g. due to privacy concerns. Finally, the open-source package described in Chapter 5 can lower the barrier for deep learning practitioners to make use of approximate inference techniques in their workflows, and serve as a building block for researchers to develop more complex methods in the future.

Contents

List of Figures

List of Tables

Listings

Chapter 1

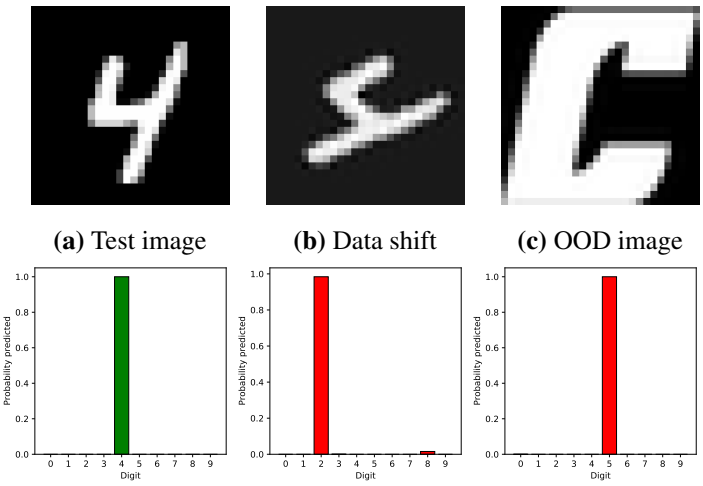
Introduction

Deep neural networks [195, 98, 304] have found wide-ranging success in learning from high-dimensional, large-scale datasets in various domains, making them the arguably most popular class of machine learning models at this time. Applications range from classical ones, such as image classification [180, 114, 115], natural language processing [322, 344, 57, 37] or speech recognition [127, 101], over more recent areas such as reinforcement learning [243, 312, 8, 313, 345] and generation of images [96] and speech [342], all the way to training on data as diverse as computer programs [102], graphs [169] and tertiary protein structures [306].

Despite their impressive performance in terms of accuracy, neural networks suffer from various problems that limit their practical applicability. One such issue is that of mis-calibration, i.e. the predicted probabilities of a class being true not corresponding to its empirical frequency [105]. In particular, neural networks tend to be overconfident in their predictions and typically assign a probability of 1 to the class that they predict as most likely. This problem is often exacerbated by shifts in the data distribution, e.g. noise or corruption of input images [120].

We visualise this phenomenon after training on an example dataset of handwritten digits [194] as in Fig. 1.1 in Fig. 1.2. While the prediction on an initial test data point (Fig. 1.2a) is correct (Fig. 1.2d), a small rotation of the image (Fig. 1.2b) leads to a near-certain incorrect prediction (Fig. 1.2e). Similarly, an image of a letter rather than a digit (Fig. 1.2c) will be assigned a class label with high confidence (Fig. 1.2e). Neural networks are further susceptible to adversarial attacks [97, 325], where an

Figure 1.1: Samples of training images for a neural classifier with two fully connected hidden layers of 512 units and ReLU non-linearities.



(a) Test image **(b)** Data shift **(c)** OOD image

(d) Test prediction **(e)** Shift prediction **(f)** OOD prediction

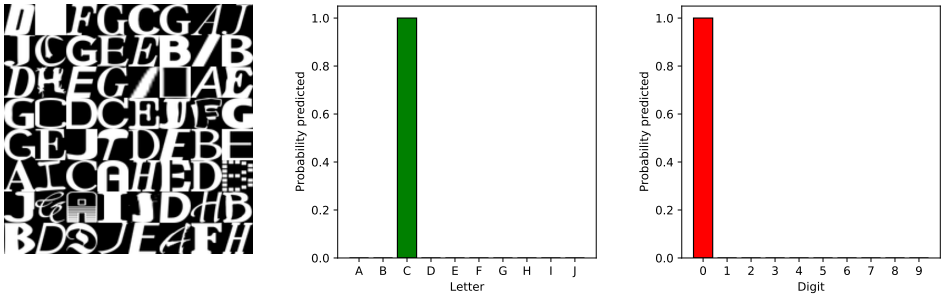
Figure 1.2: Predictions on an example image from the test set, a rotated version of the same picture and an out-of-distribution (OOD) image of a letter. While the original prediction is correct, the rotated digit is mis-classified with high confidence and a digit class is assigned with high confidence to the image of the letter.

imperceptible, targeted change to the input data drastically changes the prediction of the network.

Clearly, if we have no control over the data that is presented to a neural network at test time, we cannot trust its predictions. This is particularly problematic for safety-critical applications such as self-driving cars [43, 27] or medical imaging [207], where basing downstream decisions on incorrect predictions can have severe consequences.

Another problem that neural networks encounter is that of catastrophic forgetting, which had been studied in the early literature on neural networks [77, 232] and has received attention again more recently [95, 170]. Here, a network will 'forget' what it learned from previous data when trained on new data in an unconstrained manner, as the parameters quickly move away from old values. We show an example of this in Fig. 1.3. After having trained on the images of digits from Fig. 1.1, we now train the same network on images of the first ten letters of the alphabet as in Fig. 1.3a. While the network correctly identifies the letter from Fig. 1.2c as in Fig. 1.3b, it now makes an incorrect prediction on the digit from Fig. 1.2a as in Fig. 1.3c.

The above example is an instance of multi-task learning [42], where we aim to



(a) Second dataset. **(b)** New Fig. 1.2c prediction. **(c)** New Fig. 1.2a prediction.

Figure 1.3: After re-training the image classifier from Fig. 1.2 on letters, it is now able classify letters, but forgets to recognise the digits on which is had previously been trained.

train a single network to solve multiple unrelated tasks. However, there are many other settings in which we might want to process data as it arrives. There might simply be more data arriving over time, however the cost of training grows with the size of the dataset, so ideally we want to only use the most recent data. Similarly, we might want to transfer knowledge from one dataset to another [368] or might be in a federated learning setting [234] where we have access to small, separate datasets from which we wish to aggregate information without being able to collect all of the data in a central place. See also [108] for a recent discussion of continual learning for neural networks.

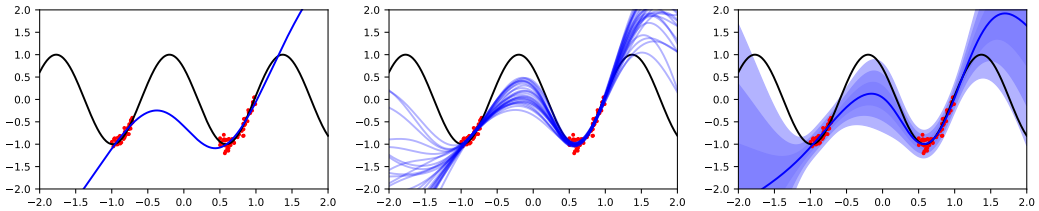
All of these issues and settings can naturally be approached as problems of Bayesian inference [19, 187, 302]. Rather than minimising a loss function to obtain parameters for a deterministic input-output mapping, a BNNs maintains uncertainty over the parameters $\boldsymbol{\theta}$ given the data $\mathcal{D}$ via the posterior distribution

$$p(\boldsymbol{\theta} \,|\, \mathcal{D}) = \frac{p(\mathcal{D} \,|\, \boldsymbol{\theta})\, p(\boldsymbol{\theta})}{p(\mathcal{D})}, \tag{1.1}$$

where $p(\mathcal{D} \,|\, \boldsymbol{\theta})$ is the likelihood, $p(\boldsymbol{\theta})$ the prior and $p(\mathcal{D}) = \int p(\mathcal{D} \,|\, \boldsymbol{\theta})\, p(\boldsymbol{\theta}) d\boldsymbol{\theta}$ the marginal likelihood or model evidence.

The posterior distribution allows us to take different parameter settings into account when predicting, which are similar around training data but quickly differ significantly away from data.

Fig. 1.4 visualises this in a $1d$ regression setting. A network trained via maxi-



(a) Maximum likelihood **(b)** HMC samples **(c)** HMC uncertainty

Figure 1.4: Toy regression example. Training a deterministic network with maximum likelihood gives us a single fit to the data, while Bayesian inference takes all possible parameter settings into account according to how well they fit the data. While highly probable parameter settings under the posterior correspond to similar functions near the training data, they extra- and interpolate differently, giving us a measure of predictive uncertainty. Ground truth function in black, predicted function in blue and noisy training data in red. Each shaded area corresponds to one additional standard deviation around the mean of the predictive posterior.

mum likelihood gives us a deterministic function as in Fig. 1.4a without any measure of uncertainty. The different functions that have a high posterior probability as in Fig. 1.4b are similar around the training data (as they need to assign a relatively high likelihood to those data points), but differ substantially in how they inter- and extrapolate. This can be summarised as a measure of the predictive uncertainty, e.g. the standard deviation of the predictions as in Fig. 1.4c.

In continual learning, the posterior always serves as the prior for the newly arriving data and succinctly summarises the philosophy behind Bayesian learning: continuously updating beliefs about the unknown in light of new information.

Unfortunately, the exact posterior is not available in closed form due to the non-linear nature of neural networks, so either closed-form approximations or sampling-based approaches are needed. Bayesian inference has first been pioneered in the context of neural networks in [220, 250]. Both methods were designed with the small-scale architectures of the time in mind and are computationally prohibitive for modern networks and datasets. In response, more recent works [100, 124, 26] have considered strong approximations to remain tractable, at the cost of restrictive factorisation assumptions in the posterior. Yet, even under a fully factorised Gaussian approximate posterior, the number of parameters doubles compared to a deterministic network, which can already be problematic in resource-constrained settings.

This thesis is broadly composed of three independent parts, the first one developing a scalable posterior approximation that can account for parameter correlations, the second one parameter-efficient variational methods, and finally the third one discussing software design considerations for flexibly implementing Bayesian approaches with modern probabilistic programming languages. More specifically, the structure is as follows:

Chapter 2 further elaborates on the techniques brought up in this introduction and lays out the background on neural networks and approximate Bayesian inference in an attempt to make this document self-contained.

Chapter 3 develops a Laplace approximation that can be applied to large neural network architectures based on structured approximations of the Hessian that account for parameter correlations without sacrificing computational efficiency, and extends the approximation to the continual learning setting. The chapter is based on the following publications:

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

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

H. Ritter, A. Botev and D. Barber. Online structured Laplace approximations for overcoming catastrophic forgetting. In *NeurIPS*, 2018.

We have recently extended this line of work to sequences of few-shot learning problems in:

P. Yap, H. Ritter and D. Barber. Addressing catastrophic forgetting in few-shot problems. In *ICML*, 2021.

However the content of this paper will not be covered in the present thesis.

Chapter 4 discusses two approaches to parameter-efficient variational inference, one through the use of binary neural networks, i.e. Bernoulli priors and variational posteriors, the other, taking inspiration from sparse Gaussian processes,

through augmenting Gaussian priors on the weight matrices with a smaller, inducing weight matrix and performing inference in that lower dimensional space.

The first half of the chapter is based on unpublished material, the second half has been published as:

H. Ritter, M. Kukla, C. Zhang and Y. Li. Sparse uncertainty representation in deep learning with inducing weights. In *NeurIPS*, 2021.

Chapter 5 presents TyXe [287], a software library at the intersection of PyTorch [269] and Pyro [22] to facilitate the use of BNNs. The corresponding paper will be published as:

H. Ritter and T. Karaletsos. TyXe: Pyro-based Bayesian neural nets for PyTorch. In *MLSys*, 2022.

Additionally, the following paper has been published during my PhD, but won't be covered in this thesis:

J. Kunze, L. Kirsch, H. Ritter and D. Barber. Gaussian mean field regularizes by limiting learned information. *Entropy*, 2019.

Chapter 2

Background

Assuming a general background in multivariate calculus, linear algebra and probability theory, this chapter introduces the necessary notation, terminology and background on neural networks, Bayesian inference and their evaluation based on uncertainty estimation for the methods developed in this thesis. The fundamental literature will be discussed here, while more closely related recent work will be covered in the context of the respective method in the corresponding sections. As this thesis is composed of multiple pieces of relatively independent work, the most relevant background material will also briefly be recapped in each section.

2.1 Neural networks

Research on neural networks dates back to 1950s and the 'Perceptron' [292, 293], a binary classification algorithm that was able to automatically update parameters based on examples and provide a threshold function based on some input features. After an initial wave of optimism that perceptrons could soon be developed into systems resembling an 'artificial intelligence', results on limitations of their expressiveness discouraged more work [239, 240]. This line of work was re-popularised in the 80s [139], in particular as multi-layer non-linear models that learn a hierarchy of internal representations of their input data, and whose parameters are updated based on gradients of an error function w.r.t. to those parameters [206, 356, 268, 294, 295]. Again, these more complex models fell out of favour, this time for more tractable convex methods [50].

Breakthroughs on learning representations of images [129] and image classification [180] have sparked the most recent wave of enthusiasm for neural networks as a class of machine learning algorithms. With progress on regularisation techniques [317], non-linearities [247], parameter initialisation [92, 113] and optimisation [166, 149] in conjunction with parallel compute devices and easy-to-use software packages [328, 45, 1, 269] becoming widely available, neural networks have established themselves as arguably the most popular item in the machine learning toolbox.

2.1.1 Architectures

On a high level, neural networks are parametric function approximations that are formed by composing simpler building blocks or 'layers'. These recursively transform some input data through intermediate representations into an output. This thesis focuses on feedforward architectures for labelled i.i.d. data, mostly images. In the following, we will give an overview of the most relevant neural network architectures, namely fully-connected and convolutional networks.

2.1.1.1 Fully-connected networks

The simplest, yet most fundamental architecture is the fully-connected network. Its core component is the fully-connected or linear layer, which calculates multiple weighted linear combinations of its inputs, so effectively corresponds to a multivariate regression model. A neural network general has at least two subsequent layers of this kind, each of which is alternated with an elementwise non-linear function.

Mathematically, a fully-connected neural network is defined as recursively mapping an input $\boldsymbol{x} =: \mathbf{a}_0$ to an output $\mathbf{a}_L$ as

$$\mathbf{h}_l = \boldsymbol{W}_l \mathbf{a}_{l-1} \tag{2.1}$$

$$\mathbf{a}_l = f_l(\mathbf{h}_l) \tag{2.2}$$

for $l = 1, \ldots, L$. $\boldsymbol{W}_l$ are the weights of layer l and f_l the non-linearity, e.g. tanh or relu where $\text{relu}(\mathbf{x})_i = \max(\mathrm{x}_i, 0)$. We refer to $\mathbf{h}$ as the pre-activations and to $\mathbf{a}$ as the activations. Commonly used bias terms can be incorporated by appending a constant

1 to all activations, i.e. defining $\mathbf{a}_0 = [\mathbf{x}^\top, 1]^\top$ and $\mathbf{a}_l = [f_l(\mathbf{h}_l)^\top, 1]^\top$. Generally, we assume that $\mathbf{a}_L = \mathbf{h}_L$, i.e. that f_L is the identity function, meaning that the output of the network is an unconstrained vector or scalar. Any transformation of the network outputs, such as to probabilities for classification, can be absorbed into the error function (see Section 2.1.2).

An interesting property of fully-connected networks is that as a model class they are invariant to a permutation of their input dimensions. As the same permutation can be applied to the columns of the first-layer weight matrix, the subsequent pre-activations will be the same as before applying the permutation and the network output will be unchanged. Hence they are well-suited for datasets with heterogeneous features, but on the other hand do not have any built-in inductive biases that exploit the local structure e.g. of images, where abrupt changes in the values of neighbouring pixels typically indicate an edge.

2.1.1.2 Convolutional networks

Originally developed with inspiration from the visual cortex [80, 79] for image recognition, convolutional neural networks (CNNs) have found wide-spread use in data domains with locally structured inputs [193]. Similar to linear layers, convolutional layers calculate a weighted linear combination of inputs. In contrast to linear layers, however, they do not consider all inputs, but apply a set of weights to a sliding windows across the input. Hence they assume a local structure within the input.

Formally, a convolutional in the context of deep learning is defined as mapping a $3d$ image tensor $X : C_{in} \times H \times W$, where the three dimensions correspond to the number of channels C_{in}, height H and width W, through a $4d$ weight tensor $W : C_{out} \times C_{in} \times h \times w$ to another $3d$ tensor. The value at each location i, j in channels k is computed as

$$H_{k,i,j} = [X * W]_{k,i,j} = \sum_{\delta_h=-\lfloor h/2 \rfloor}^{\lfloor h/2 \rfloor} \sum_{\delta_w=-\lfloor w/2 \rfloor}^{\lfloor w/2 \rfloor} \sum_{c'=1}^{c_{in}} X_{c',i+\delta_h,j+\delta_w} W_{k,c',\lfloor h/2 \rfloor+\delta_h,\lfloor w/2 \rfloor+\delta_w}. \tag{2.3}$$

At its core, a convolution is a linear mapping with a sum over $K = C_{in}hw$ terms. Hence we can equivalently express it as a matrix multiplication by rearranging X

into a matrix $\boldsymbol{X} : K \times M$, where M corresponds to the number of locations in the resulting tensor H, and W into $\boldsymbol{W} : C_{out} \times C_{in}hw$

$$\boldsymbol{H} = \boldsymbol{W}\boldsymbol{X}. \tag{2.4}$$

So another view of convolutional layers is that of a linear layer with shared weights across all input patches to the layer.

Besides the shape of the weight tensor, the most commonly used hyperparameters are the padding, i.e. a constant value that is filled in around the borders of the input to preserve its size, and the stride, i.e. the shift in the center of the convolution, which can be used for reducing the size of the input.

Another component often found in convolutional networks is the pooling or sub-sampling layer. This applies a parameter-free aggregation function to a region of inputs, e.g. they might calculate the mean or the maximum over a patch of 2×2 pixels. In conjunction with convolutional and linear layers, pooling layers still find use in most modern CNNs. Some of the historically most influential networks, such as LeNet [190, 191] and AlexNet [180], were composed entirely of these.

A more recent addition to many CNN architectures is Batch Normalisation [149], which normalises its input by subtracting the mean and dividing by the standard deviation and replaces these with a learnt mean and scale

$$BN(x) = \gamma\frac{x - \mathbb{E}[x]}{\sqrt{\mathbb{V}[x]}} + \beta, \tag{2.5}$$

where β is the learnt mean and γ the scale. The empirical mean and variance here are calculated from mini-batch statistics during training, and running averages accumulated throughout training for test-time prediction. There is currently no formal understanding of how and when Batch Normalisation indeed leads to improvements, but the consensus in the literature revolves around it simplifying the loss landscape and thus aiding optimisation [25, 301].

Finally, 'residual layers' have recently been popularised by ResNets [114, 115]. They pass their input $\mathbf{x}$ through a block g consisting of a sequence of the previously

discussed neural network components and then adds the result to the input

$$r(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}). \tag{2.6}$$

If g is initialised to map all inputs to (almost) zero, the residual block is overall initialised to the identity, which has proven itself as a helpful inductive bias in neural network training. In ResNets, g consists of a small number of convolutional, Batch Normalisation and non-linearity layers. These residual blocks are then repeated at increasing channel sizes to form one of the most widely used architectures for image data at this time.

2.1.2 Training

2.1.2.1 Loss functions

In supervised learning, which is the focus of this thesis, neural networks are generally trained by finding some parameter values $\boldsymbol{\theta}^*$ that minimise a loss function, i.e.

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}), \tag{2.7}$$

where the loss is defined as a sum of per-datapoint errors

$$\mathcal{L}(\boldsymbol{\theta}) = E_{\boldsymbol{\theta}}(\mathcal{D}) = \sum_{i}^{N} E(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = N\,\mathbb{E}_{\mathbf{x},y\sim p(\mathcal{D})}\left[E(f_{\boldsymbol{\theta}}(\mathbf{x}), y)\right], \tag{2.8}$$

and $\mathcal{D} = \{(\mathbf{x}_i, y)\}_{i=1}^{N}$ is the dataset consisting of N inputs $\mathbf{x}$ and labels y.

Examples of commonly used error functions are the squared error for regression where $y \in \mathbb{R}$

$$E(f(\mathbf{x}), y) = (f(\mathbf{x}) - y)^2, \tag{2.9}$$

binary cross-entropy for binary classification where $y \in \{0, 1\}$

$$E(f(\mathbf{x}), y) = y \log \sigma(f(\mathbf{x})) + (1 - y) \log(1 - \sigma(f(\mathbf{x}))), \tag{2.10}$$

and $\sigma(a) = (1 + e^{-a})^{-1}$ is the sigmoid or logistic function. Finally, for multi-class

classification, where y is a D-dimensional vector of all 0s except for a single 1, the cross entropy loss is:

$$E(f(\mathbf{x}), \mathbf{y}) = -\sum_{d=1}^{D} \mathrm{y}_d \log \mathrm{softmax}(f(\mathbf{x}))_d, \tag{2.11}$$

where $\mathrm{softmax}(\mathbf{a})_d = e^{\mathrm{a}_d} / \sum_{d'} e^{\mathrm{a}_{d'}}$ is the softmax function that maps a real-valued vector to the probability simplex.

Often a regularisation term that only depends on the parameters is further added to the data-dependent error term

$$\mathcal{L}(\boldsymbol{\theta}) = E_{\boldsymbol{\theta}}(\mathcal{D}) + \frac{\lambda}{2}\mathcal{R}(\boldsymbol{\theta}), \tag{2.12}$$

where λ is a scalar hyperparameter that determines the relative weight of the regularisation term. This is to avoid overfitting, i.e. adapting the parameters to noise rather than signal in the training data that does not generalise to independent test data.

The most commonly used regulariser is the squared error or L2-norm

$$\mathcal{R}(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_2^2 = \sum_i \theta_i^2. \tag{2.13}$$

2.1.2.2 Gradients and automatic differentiation

Due to the non-linear nature of neural networks, the loss function cannot be minimised w.r.t. the parameters in closed form. However, we can find a (local) minimum via gradient descent. That is, we choose some initial values $\boldsymbol{\theta}_0$ for the parameters, typically through random initialisation, and update these values iteratively as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t), \tag{2.14}$$

using the gradients of the loss. While the gradients of the regulariser are typically easy to determine, the procedure for the data error is somewhat more involved. Yet, these can be calculated by recursively applying the chain rule, i.e. 'backpropagation' [294, 295] or reverse-mode automatic differentiation. For example, the gradients w.r.t. the

weights of some layer l are

$$\nabla_{\boldsymbol{W}_l} E(\boldsymbol{\theta}) = \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{W}_l} = \frac{\partial \mathbf{h}_l}{\partial \boldsymbol{W}_l} \frac{\partial \mathbf{a}_l}{\partial \mathbf{h}_l} \frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{a}_l} = \mathbf{a}_{l-1}^\top \operatorname{diag}(f_l'(\mathbf{h}_l)) \frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{a}_l}, \tag{2.15}$$

where $\frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{a}_l}$ can be calculated via the chain rule and the recursion is initialised with the gradient of the loss w.r.t. the network outputs.

2.1.2.3 Stochastic gradient descent

The core realisation towards scaling neural network training to arbitrarily large datasets, is that we do not need to calculate gradients over the full dataset, but can instead calculate them over random subsets of data and optimise using *stochastic* gradient descent [291, 30, 31].

Since the error function is an expectation over the data, which does not depend on the parameters, our gradients are also an expected value

$$\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}) = N \nabla_{\boldsymbol{\theta}} \mathbb{E}_{p(\mathbf{x},y)} [E(\boldsymbol{\theta})] = N \mathbb{E}_{p(\mathbf{x},y)} [\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta})], \tag{2.16}$$

where $p(\mathbf{x},y)$ is the empirical training data distribution with a point mass on each data point, i.e. $p(\mathbf{x},y) = 1/N$ if $(\mathbf{x},y) \in \mathcal{D}$ and 0 otherwise.

We can get an unbiased Monte Carlo estimate of the of the gradient by choosing a random subset $\mathcal{M} \subset \mathcal{D}$ of size M and approximate the gradient as

$$\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}) \approx \frac{N}{M} \sum_{\mathbf{x},y \in \mathcal{M}} \nabla_{\boldsymbol{\theta}} E(f_{\boldsymbol{\theta}}(\mathbf{x}), y). \tag{2.17}$$

In practice, a common approach is to sample the data points without replacement, i.e. iterate over the dataset in random order for an entire pass or 'epoch', before returning to a data point.

Typically, rather than using a fixed step size η, it is decayed over time, e.g. exponentially as $\eta_t = c^t \eta_0$ with $0 < c < 1$, or kept constant for some number of iterations and then reduced after some number of updates, e.g. as $\eta_t = c_1$ if $0 < t \leq N_0$, $\eta_t = c_2$ if $N_0 < t \leq N_1$ etc.

Gradient descent as described above can be slow to converge, since each update

is calculated independently and may be too small to make sufficient progress in flat regions of the loss landscape. A popular improvement is to maintain a momentum term or velocity $\mathbf{v}$ for each parameter, which makes the optimisation procedure more similar to a ball rolling down a surface [273]. The velocity and parameters are then updated as

$$\mathbf{v}_t = \mu \mathbf{v}_{t-1} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{t-1}) \tag{2.18}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} + \mathbf{v}_t, \tag{2.19}$$

where μ is an additional momentum hyperparameter determining the relative weight of the newest gradient estimate against the history of gradients.

A slightly more involved approach is Nesterov momentum [252], which updates the parameters and velocity as

$$\mathbf{v}_t = \mu \mathbf{v}_{t-1} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{t-1}) \tag{2.20}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mu \mathbf{v}_{t-1} + (1+\mu)\mathbf{v}_t, \tag{2.21}$$

which provably accelerates convergence for convex objectives [252].

Yet, using a scalar learning rate, i.e. the same rate for all parameters, can be overly simplistic for models with error surfaces as complex as neural networks. In recent years, adaptive gradient methods, which automatically set a learning rate for each parameters, have attracted significant attention [61, 370, 128]. Of these, Adam [166] has emerged as arguably the most popular one, and despite some concerns regarding its behaviour around minima [286], often gives good performance without significant tuning of learning rates or decay schedules. Specifically, Adam maintains moving averages of the gradients and squared gradients, and updates the

parameters as

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{t-1}) \tag{2.22}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{t-1})^2 \tag{2.23}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \frac{\eta}{\sqrt{\mathbf{v}_t/(1-\beta_2^t)} + \varepsilon} \odot \frac{\mathbf{m}_t}{1-\beta_1^t}, \tag{2.24}$$

where β_1, β_2 are momentum hyperparameters, the ε is a small positive constant to avoid division by zero, the square operation is elementwise and $\odot$ denotes elementwise multiplication of two vectors. Finally, the division of the gradient and squared-gradient moving averages in the parameter-update equation is to account for them being biased towards zero through their initialisation. Scaling the updates by the inverse of the squared gradients allows for large steps being taken in directions with small gradients and small ones in directions with large gradients, enabling fast progress of the optimisation procedure while reducing the risk of taking overly large steps that lead to divergence.

2.1.3 Probabilistic interpretation

Neural networks can also be interpreted from a probabilistic viewpoint. Here, we typically assume independence between the data given the parameters and have the output of the network for the inputs parameterise the corresponding likelihood of the label. The (unregularised) error function used for optimisation is then the negative log likelihood of the data

$$E(\boldsymbol{\theta}) = -\log p(\mathcal{D} \,|\, \boldsymbol{\theta}) = -\sum_i \log p(y_i \,|\, f_{\boldsymbol{\theta}}(\mathbf{x})). \tag{2.25}$$

Most commonly used error function, such as those discussed in the previous section, have corresponding data likelihoods, so that unregularised neural network training can be interpreted as maximum likelihood estimation in the probabilistic model. For the squared error in regression, the corresponding likelihood is the

Gaussian distribution

$$p(y \mid \mathbf{x}, \boldsymbol{\theta})) = \mathcal{N}(y; f_{\boldsymbol{\theta}}(\mathbf{x}), \tau^{-1}), \tag{2.26}$$

with the network outputs corresponding to the mean of the Gaussian in label space. The variance τ may be known or can be learnt, but only plays a role when we place a prior on the parameters for a full Bayesian treatment that can also account for regularisation (see next section).

For binary classification, we have a Bernoulli likelihood with

$$p(y = 1 \mid \mathbf{x}, \boldsymbol{\theta})) = \sigma(f_{\boldsymbol{\theta}}(\mathbf{x})). \tag{2.27}$$

And finally for multi-class classification, the corresponding likelihood is the Categorical distribution

$$p(\mathrm{y}_d = 1 \mid \mathbf{x}, \boldsymbol{\theta})) = \mathrm{softmax}(f_{\boldsymbol{\theta}}(\mathbf{x}))_d. \tag{2.28}$$

In both classification settings the (unconstrained) network outputs correspond to the logits of the classes.

2.2 Bayesian inference

We will now move to a full Bayesian treatment of neural networks and performing inference in them. Bayesian inference is of course much more broadly applicable than in neural network models, see e.g. [223, 24, 17, 244, 88] for introductory texts. The key step from a probabilistic view of training deterministic neural networks to Bayesian deep learning is placing a prior $p(\boldsymbol{\theta})$ on the parameters in addition to interpreting them as parameterising a likelihood. Through Bayes' rule, the combination of the sum and product rules of probability, we then obtain the posterior as

$$\underbrace{p(\boldsymbol{\theta} \mid \mathcal{D})}_{\text{Posterior}} = \frac{\overbrace{p(\mathcal{D} \mid \boldsymbol{\theta})}^{\text{Likelihood}} \overbrace{p(\boldsymbol{\theta})}^{\text{Prior}}}{\underbrace{p(\mathcal{D})}_{\text{Evidence}}} = \frac{p(\mathcal{D} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta})}{\int p(\mathcal{D} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta}) d\boldsymbol{\theta}} \propto p(\mathcal{D} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta}). \tag{2.29}$$

When making predictions on unseen data, the posterior is used to average over all possible parameter settings to find the predictive posterior

$$p\left(y' \,\middle|\, \mathbf{x}', \mathcal{D}\right) = \mathbb{E}_{p(\boldsymbol{\theta}\,|\,\mathcal{D})}\left[p\left(y' \,\middle|\, \mathbf{x}', \boldsymbol{\theta}\right)\right] = \int p\left(y' \,\middle|\, \mathbf{x}', \boldsymbol{\theta}\right) p\left(\boldsymbol{\theta} \,|\, \mathcal{D}\right) d\boldsymbol{\theta}. \tag{2.30}$$

Since this expectation is generally not available in closed form, it is usually estimated through a Monte Carlo approximation with T samples

$$p\left(y' \,\middle|\, \mathbf{x}', \mathcal{D}\right) \approx \frac{1}{T} \sum_{t=1}^{T} p\left(y' \,\middle|\, \mathbf{x}', \boldsymbol{\theta}_t\right) \qquad \text{with} \qquad \boldsymbol{\theta}_t \sim p\left(\boldsymbol{\theta} \,|\, \mathcal{D}\right). \tag{2.31}$$

The main difficulty in evaluating the posterior is the evidence or marginal likelihood as this involves an integral over the full parameter space. As this term is a constant w.r.t. the parameters, however, finding a mode of the posterior is no more difficult than the deterministic optimisation problems discussed in the previous section. In particular, we can define a loss function for training a neural network as additively proportional to the negative log posterior

$$\mathcal{L}(\boldsymbol{\theta}) = -\log p\left(\mathcal{D} \,|\, \boldsymbol{\theta}\right) - \log p(\boldsymbol{\theta}). \tag{2.32}$$

Finding a mode of the posterior is also referred to as Maximum a-Posteriori or MAP inference and can be seen as approximating the posterior as a delta distribution with infinite mass at the value of the MAP parameters

$$p\left(\boldsymbol{\theta} \,|\, \mathcal{D}\right) \approx q_{MAP}(\boldsymbol{\theta}) := \delta(\boldsymbol{\theta} = \boldsymbol{\theta}^*) = \begin{cases} \infty & \text{if } \boldsymbol{\theta} = \boldsymbol{\theta}^* \\ 0 & \text{otherwise.} \end{cases} \tag{2.33}$$

This reduces the predictive posterior to the prediction made by the optimised parameters.

Despite its conceptual simplicity, Bayesian inference is challenging computationally as the posterior is generally not available in closed form. In many cases this is due to the structure of the probabilistic model, however in the case of neural networks the main difficulty lies in their non-linearity, exacerbated by the typically

high dimensionality of $\boldsymbol{\theta}$.

Hence, approximate inference procedures are required and particular care needs to be taken to remain computationally tractable. We will refer to any neural network where an attempt is made to approximate the posterior over the parameters as a Bayesian neural network (BNN). Below, we provide an overview of approaches for approximate inference, with a particular emphasis on the Laplace approximation and variational inference, where this thesis makes its core contributions.

2.2.1 Laplace approximation

A first approximate inference procedure that we will discuss is the Laplace or – for a more descriptive name – quadratic approximation [187]. This technique forms the basis of Chapter 3.

The core idea is that once we have found a mode or MAP estimate of the posterior as described previously, we can add uncertainty in the parameter space post-hoc based on the geometry around that mode. Specifically, the Laplace approximation exploits the connection between deterministic training with the negative log-posterior as an objective and Bayesian inference by approximating the loss as locally quadratic with a 2nd-order Taylor expansion around a minimum

$$\mathcal{L}(\boldsymbol{\theta}) \approx \underbrace{\mathcal{L}(\boldsymbol{\theta}^*)}_{\text{constant}} + (\boldsymbol{\theta}-\boldsymbol{\theta}^*)^\top \underbrace{\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^*)}_{0} + \frac{1}{2}(\boldsymbol{\theta}-\boldsymbol{\theta}^*)^\top \nabla^2_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^*)(\boldsymbol{\theta}-\boldsymbol{\theta}^*) \tag{2.34}$$

$$\simeq \frac{1}{2}(\boldsymbol{\theta}-\boldsymbol{\theta}^*)^\top \nabla^2_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^*)(\boldsymbol{\theta}-\boldsymbol{\theta}^*), \tag{2.35}$$

where $\simeq$ denotes additive proportionality, the first-order term is 0 due to expanding around a mode, and

$$\nabla^2_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^*) = \frac{\partial^2 \mathcal{L}(\boldsymbol{\theta}^*)}{\partial \boldsymbol{\theta}^2} \tag{2.36}$$

is the Hessian of the negative log posterior evaluated at the MAP parameters.

Since the posterior is proportional to the negative exponentiated loss, it is approximately proportional to

$$p(\boldsymbol{\theta}\,|\,\mathcal{D}) \propto \exp\{-\mathcal{L}(\boldsymbol{\theta})\} \approx \exp\left\{-\frac{1}{2}(\boldsymbol{\theta}-\boldsymbol{\theta}^*)^\top \nabla^2_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^*)(\boldsymbol{\theta}-\boldsymbol{\theta}^*)\right\}. \tag{2.37}$$

We can recognise this as an unnormalised Gaussian probability density with the Hessian of the negative log posterior being the precision matrix or inverse of the covariance, so that the Laplace approximation to the posterior is

$$p(\boldsymbol{\theta} \mid \mathcal{D}) \approx q_{LAP}(\boldsymbol{\theta}) := \mathcal{N}(\boldsymbol{\theta}^*, \nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^*)^{-1}). \tag{2.38}$$

Note that, by default, the Laplace approximation uses the inverse of the full Hessian as the covariance. This means that in space complexity it scales as $\mathcal{O}(d^2)$ and $\mathcal{O}(d^3)$ in time complexity for calculating the (square root of the) inverse, where $d = |\boldsymbol{\theta}|$ is the total number of parameters. Hence its use is prohibitively expensive for models with a large number of parameters, see Chapter 3 for specific numerical examples with neural networks.

The Laplace approximation bases its approximation of the posterior on a quadratic approximation of the loss around a mode. From a practical point of view this two-stage procedure of first optimising deterministic parameters and only adding uncertainty over the parameters post-hoc is highly appealing, as it allows for 'Bayesianising' pre-existing models without any modifications to an existing training pipeline. However, the accuracy of the posterior approximation depends of course on the optimisation procedure finding a representative mode (for multi-model posteriors) and how well the log posterior is locally described by a quadratic. Higher-order effects and asymmetries can lead to the Laplace approximation over- or under-assigning probability mass to areas in parameter space [255], as it only takes the local curvature information around a single point, the MAP estimate, into account.

A common justification for the Laplace approximation is the Bernstein-von-Mises theorem [189, 347, 343], which, informally, states that the posterior in an identifiable model converges in the infinite data limit to a Normal distribution around the maximum likelihood estimate with covariance equal to the inverse Fisher information (which for now can be thought of as equal to the Hessian and will be defined later) divided by the number of data. Unfortunately, neural networks are unidentifiable (e.g. they are invariant to a permutation of the hidden units in fully

connected networks and a permutation of the channels in convolutional networks) and the Fisher information is singular [349, 350]. Therefore, neural network posteriors do not enjoy these theoretical guarantees. Even if the theory applied, in most practical settings the number of parameters greatly exceeds the number of data, hence even for the most generous interpretation of approaching infinity, neural networks would be far off the asymptotic regime. Yet, as Radford Neal already noted in his PhD thesis during the previous iteration of the deep learning hype cycle: *"to hesitate because of such qualms would be contrary to the spirit of the neural network field"* [250].

2.2.2 Variational inference

To explicitly account for matching the mass of the posterior, variational inference [346, 372] introduces a parametric family approximation to posterior, e.g. a Gaussian, and optimises its parameters such that the parametric approximation is similar. Specifically it optimises a lower bound on the marginal likelihood, the evidence lower bound (ELBO), which can be derived as minimising the KL divergence between the parametric approximation $q_{\boldsymbol{\phi}}$ and $p(\boldsymbol{\theta} \mid \mathcal{D})$

$$\begin{aligned}
\mathbb{KL}[q(\boldsymbol{\theta}) \,||\, p(\boldsymbol{\theta} \mid \mathcal{D})] &= \mathbb{E}_{q(\boldsymbol{\theta})}\left[\log \frac{q(\boldsymbol{\theta})}{p(\boldsymbol{\theta} \mid \mathcal{D})}\right] && (2.39)\\
&= \mathbb{E}_{q(\boldsymbol{\theta})}\left[\log \frac{q(\boldsymbol{\theta}) p(\mathcal{D})}{p(\mathcal{D} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta})}\right] && (2.40)\\
&= \log p(\mathcal{D}) - \mathbb{E}_{q(\boldsymbol{\theta})}\left[\log \frac{p(\mathcal{D} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta})}{q(\boldsymbol{\theta})}\right] && (2.41)\\
&= \log p(\mathcal{D}) - \underbrace{(\mathbb{E}_{q(\boldsymbol{\theta})}\left[\log p(\mathcal{D} \mid \boldsymbol{\theta})\right] - \mathbb{KL}[q(\boldsymbol{\theta}) \,||\, p(\boldsymbol{\theta})])}_{\text{ELBO}} && (2.42)
\end{aligned}$$

$$\Leftrightarrow \text{ELBO}(q(\boldsymbol{\theta})) = -\mathcal{L}(q(\boldsymbol{\theta})) = \log p(\mathcal{D}) - \mathbb{KL}[q(\boldsymbol{\theta}) \,||\, p(\boldsymbol{\theta} \mid \mathcal{D})] \leq \log p(\mathcal{D}), \tag{2.43}$$

so that optimising the ELBO reduces the mis-match as measured by the KL divergence between the approximate and the true posterior. Crucially, the difficult integration problem for finding $p(\mathcal{D})$ has now been turned into an optimisation problem for finding parameter of an integral with a known solution – $q_{\boldsymbol{\phi}}$ – that matches

the integral with unknown solution as closely as possible. A simple and efficient choice for $q_{\boldsymbol{\phi}}$ for inference over unconstrained real-valued variables such as neural network weights is a factorised Gaussian, i.e.

$$q_{\boldsymbol{\phi}}(\boldsymbol{\theta}) = \prod_i \mathcal{N}(\theta_i; \mu_i, \sigma_i^2) = \mathcal{N}(\boldsymbol{\theta}; \boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2)), \tag{2.44}$$

s.t. $\boldsymbol{\phi} = [\boldsymbol{\mu}^\top, \boldsymbol{\sigma}^\top]^\top$, which means that for every variable (or deterministic parameter with ML or MAP estimation) we now have two parameters, a mean and a variance for a univariate Gaussian over that parameter. This approximation has been used in multiple variational inference approaches for BNNs [130, 100, 26, 168, 164, 260, 361, 324].

While early approaches typically involved deriving model-specific update equations [14], recent work has shifted towards developing black-box methods for arbitrary arbitrary models based on stochastic optimisation. For estimating the log likelihood term, mini-batches can be utilised for both local and global variables [136]. Gradients w.r.t. $\boldsymbol{\phi}$ can further be estimated by sampling from $q_{\boldsymbol{\phi}}$. This can be done for arbitrary distributions that allow for efficient sampling [278, 182] via the score-function trick [357]. More recently, it has been observed that 'reparameterisation gradients'[167, 332], i.e. samples that are differentiable w.r.t. the parameters of their distribution, reduce the variance of the gradients and improve optimisation. This applies e.g. for our Gaussian example above as

$$\boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2)) \;\Leftrightarrow\; \boldsymbol{\theta} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\varepsilon} \quad \text{with} \quad \boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}) \tag{2.45}$$

is differentiable w.r.t. both $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$, hence gradients w.r.t. the log likelihood can be estimated efficiently if the log likelihood is differentiable w.r.t. $\boldsymbol{\theta}$. For neural networks specifically, it can also be feasible to integrate out the distributions over the parameters through deterministic computations such as the linear mapping of a fully connected layer and obtain distributions over the respective outputs [168]. If those distributions allow for reparameterisable sampling, sampling from these can allow for estimating the gradients with even lower variance.

The specific choice of the parametric family for $q_{\boldsymbol{\phi}}$ is generally a trade-off between computational efficiency, which encourages simpler, factorised distributions, and expressivity to allow for the approximation to more closely match the true posterior, as the ELBO is maximised by the posterior and only a family containing it can achieve this maximum. Interestingly, however, more expressive variational families do not necessarily lead to better predictive performance [339].

2.2.3 Markov chain Monte Carlo

While the present thesis does not make any contribution in this direction, we will briefly cover Markov chain Monte Carlo (MCMC) methods here for the sake of completeness. In contrast to the Laplace approximation and variational inference, MCMC avoids defining a closed-form approximation to the posterior and instead relies on drawing a sequence of samples that (in the infinite limit) are from the posterior distribution as e.g. for approximating the predictive posterior only samples are required. Most commonly used approaches were derived from the Metropolis-Hastings algorithm [236, 110], which picks some transition probabilities $p\left(\boldsymbol{\theta}' \,\middle|\, \boldsymbol{\theta}\right)$ (such that the sequence of samples form a Markov chain) and then accepts a sample with probability

$$A = \min\left(1, \frac{p\left(\boldsymbol{\theta}' \,\middle|\, \mathcal{D}\right) p\left(\boldsymbol{\theta} \,\middle|\, \boldsymbol{\theta}'\right)}{p\left(\boldsymbol{\theta} \,\middle|\, \mathcal{D}\right) p\left(\boldsymbol{\theta}' \,\middle|\, \boldsymbol{\theta}\right)}\right). \tag{2.46}$$

While the posterior probability of a single setting of the parameters can generally not be computed, in the odds of two different settings the normaliser cancels out, making the acceptance probability of a transition tractable.

As MCMC methods are asymptotically exact, they are generally considered the 'gold standard' for inference. In particular HMC [250, 249, 251], further developed as NUTS [135], which proposes transitions based on gradients of the negative log posterior and a random momentum term, has enjoyed tremendous success on small datasets with neural network models. However, as calculating the acceptance probability of a transition requires evaluating the full log likelihood, these methods are not practical for modern-scale datasets and architectures and they are mainly used as a ground-truth baseline (although recent work [152] has investigated HMC

on some large-scale settings). As the accept-reject step requires evaluating the likelihood of the full dataset, stochastic gradient methods that take sufficiently small steps to assume that the acceptance probability is 1 have received attention [351, 4, 202, 44, 219, 117, 374].

2.2.4 Ad-hoc methods

Finally, various ad-hoc methods that do not explicitly approximate the posterior, but construct a distribution over parameters and hence give access to the results of different forward passes for averaging, have become popular in the literature due to their simplicity and practicality. In fact, recent work argues that model averaging is the core component of Bayesian deep learning [358, 359]. As some of these methods will be considered as baselines in this work, we will give an overview of the most relevant ones.

2.2.4.1 Deep ensembles

Most notably deep ensembles [185] have found wide-ranging success in not only reducing test errors over individual models, but also offering comparably strong uncertainty estimation as compared to other techniques [315]. Implementation-wise, they are as simple to construct as training multiple deterministic models independently from different intitialisations of the parameters. The distribution over the parameters can then be regarded as a mixture over K delta distributions (for an ensemble of K networks)

$$q_{ENS}(\boldsymbol{\theta}) = \frac{1}{K}\sum_{k=1}^{K}\delta(\boldsymbol{\theta} = \boldsymbol{\theta}_k). \tag{2.47}$$

However, deep ensembles are a computationally expensive method, requiring both storing K times as many parameters as well as performing that many forward passes through the network. Further, as they rely on training multiple ML or MAP models, with little data or limited diversity thereof, trained networks may end up making the same predictions [263], hence taking a principled approach with a faithful approximation of the posterior promises to be more robust across different settings.

2.2.4.2 Monte Carlo Dropout

Various stochastic regularisation techniques for neural networks have been interpreted as performing approximate inference, in particular Dropout [83] and Batch Normalisation [327, 12]. Dropout has originally been proposed as a regularisation technique where individual units of the network, i.e. dimensions of the activations, are randomly set to zero in the forward pass during training. This can also been formulated as sampling the weights of a layer as

$$\boldsymbol{W}_l = \boldsymbol{M}_l \operatorname{diag}(\mathbf{z}_l) \quad \text{with} \quad \mathbf{z}_l \sim \mathcal{B}(\mathbf{p}_l), \tag{2.48}$$

where $\boldsymbol{M}$ is a learnable parameters that takes the role of the weight matrix in deterministic networks and $\mathcal{B}(\mathrm{p})$ is the Bernoulli distribution with probability p for a positive outcome. For testing, the usual approach is to set the weights deterministically as

$$\boldsymbol{W}_l = \boldsymbol{M}_l \operatorname{diag}(\mathbf{p}_l) \tag{2.49}$$

to maintain the expected magnitude of the linear mapping. In contrast, MC Dropout [83] proposes averaging the outputs of multiple stochastic forward passes as in the Bayesian model average and argues that training a network with Dropout is a form of variational inference. However, concerns have been raised both that this interpretation [140, 141] is not sound and that the predictive uncertainty estimates can be poorly calibrated [263, 73]. An alternative interpretation as a prior [248] has been put forward. Nevertheless, MC Dropout is frequently used in practical settings [157, 200, 246] as Dropout is a commonly used building block in many architectures and hence MC dropout gives access to improved uncertainty estimates over deterministic networks with minimal overhead.

2.3 Evaluation: uncertainty estimation

As the true posterior is intractable, approximate inference techniques for BNNs can generally not be evaluated as to how well they approximate the posterior outside of toy settings. Further, accurate sampling techniques such as HMC are only available

at a significant computational expense for larger datasets and architectures [152]. Hence, besides commonly used supervised measures on the predictive mean, such as squared or classification error/accuracy and negative log likelihood (NLL), the empirical evaluation of BNNs usually revolves around downstream uncertainty estimation tasks (typically in classification settings) on which deterministic networks tend to perform poorly.

2.3.1 Calibration

One such example is that of calibration on the test set. Deterministic networks tend to exhibit overconfidence [105], i.e. they assign a higher probability to their prediction than their empirical accuracy. The most commonly used measure of calibration is the expected calibration error, which for a binary classifier is calculated by grouping the predicted probabilities into B bins of equal width and calculating

$$ECE = \sum_b \frac{n_b}{N} \left| acc(b) - conf(b) \right|, \tag{2.50}$$

where n_b denotes the number of predictions assigned to bin b, $acc(b)$ their accuracy and $conf(b)$ their confidence, i.e. their average value. For multi-class problems we can take the highest predicted probability for each class or treat all class predictions independently. Further variants, such as using bins of equal size rather than fixed width, have recently been discussed in [257]. In any case, the ECE should never be used as the sole measure of the quality of a classifier, as input-independent solutions such as predicting the probability for each class as its empirical frequency trivially optimise the ECE. Hence, it is mostly useful for comparing classifiers with similar accuracy.

For a single calibration measure, recent work [315] advocates for the use of proper scoring rules [93, 35], which measure the accuracy of probability predictions and are uniquely optimised by the correct set of predictions. One such scoring rule is the Brier score, the squared difference between predicted probabilities and the

one-hot encoding of the label

$$Brier(\mathbf{p}) = \frac{1}{C}\sum_c (\mathrm{p}_c - \mathrm{y}_c)^2. \tag{2.51}$$

Negative log likelihoods are usually also proper scoring rules [93]. However, these all share the drawback of being a function of predictive accuracy, i.e. when one classifier outperforms another in terms of the Brier score, it is not clear whether this is due to better discriminative performance or better calibration.

2.3.2 Robustness to distributional shift

To increase the difficulty of the benchmarking problems, a common procedure is to apply perturbations to the test data, such as random noise or image distortions [120, 315] to simulate a shift of the data distributions. As the data usually remains recognisable to humans, a 'robust' model would maintain its test performance on the shifted data. Robustness and data shift have become an increasingly popular problem setting in the recent literature, leading to the development of more realistic benchmarks [174, 226].

2.3.3 Out-of-distribution detection

Another popular task is detecting out-of-distribution (OOD) inputs based on the uncertainty of the predictive distribution [121, 315]. This uncertainty is measured, for example, by the entropy, which for classification with predicted probabilities $\mathbf{p}$ is defined as

$$\mathbb{H}(\mathbf{p}) = -\sum_c \mathrm{p}_c \log \mathrm{p}_c. \tag{2.52}$$

The commonly used priors for a BNN generally lead to a uniform prediction over classes in classification, i.e. the maximum possible entropy. After observing the training data, the uncertainty near these is reduced to highly confident predictions near those data points under the posterior. Therefore, we would expect an approximate inference procedure that represents the true posterior well to exhibit low uncertainty on known data and high uncertainty on unseen data, therefore being able to distinguish between the in-distribution test data and previously unobserved OOD

data. In contrast, deterministic networks with ReLU non-linearities are provably over-confident away from the training data [118] and it can be shown that a Bayesian treatment alleviates this issue [176].

2.3.4 Adversarial attacks

A further weakness of deterministic networks is their susceptibility to adversarial attacks [97, 325], which are artificial inputs constructed to alter the predictions of the neural network. Examples include noise-like data points that receive high-confidence predictions as well as imperceptible changes to real data points that change the network's prediction to another class. Specifically, the untargeted fast gradient sign method (FGSM) [97] sets its input as

$$\mathbf{x}_{ADV} = \mathbf{x} - \eta \, \mathrm{sgn}(\nabla_{\mathbf{x}} \mathrm{p}_i), \tag{2.53}$$

with $i = \arg\max_j \mathrm{p}_j$ and η being the scalar step size. Similarly the targeted FGSM takes the gradient w.r.t. the input for a specific class but adds the gradient in order to increase the corresponding classes' probability. First examples of works observing that BNNs are more robust to adversarial attacks include [205, 285, 84, 20, 211], with [40] providing a more recent systematic study. Briefly, the argument for BNNs being more robust to adversarial attacks is that such inputs are presumed to lie off the manifold of training data and are therefore out-of-distribution data where BNNs are increasingly uncertain. Yet, adversarial examples are a sub-field that has been attracting significant attention with a wide range of attacks and defenses having been developed over the past years and FGSM is a rather simplistic attack.

2.3.5 Uncertainty and Risk

Finally, BNNs allow us to distinguish between different types of uncertainty: risk inherent in the data and uncertainty over the model [172], which can be reduced by collecting additional data. These are also referred to as 'aleatoric' and 'epistemic' uncertainty [56, 161, 144]. The distinction is possible through the likelihood of a prediction given a specific set of parameters as well as the posterior over these parameters.

In regression problems this is relatively straight-forward, as we typically have a variance for the Gaussian over the prediction. This variance may depend on the input and be predicted alongside the mean by the network in the case of heteroskedastic regression to add to the variance of the predictive means. The latter can be approximated by drawing multiple samples from the approximate posterior.

For classification, the categorical distribution lacks a similar notion of a variance parameter. However, we can leverage tools from information theory [308] such as the mutual information between the parameters and the predictions. There are various ways of calculating the mutual information, the most convenient one in the context of Bayesian deep learning [81] is to take the difference between the marginal and the average conditional entropy

$$\mathbb{MI}\left(y', \boldsymbol{\theta} \,\middle|\, \mathbf{x}', \mathcal{D}\right) = \mathbb{H}\left(p\left(y' \,\middle|\, \mathbf{x}', \mathcal{D}\right)\right) - \mathbb{E}_{p(\boldsymbol{\theta} \,|\, \mathcal{D})}\left[\mathbb{H}\left(p\left(y' \,\middle|\, \mathbf{x}', \boldsymbol{\theta}\right)\right)\right]. \tag{2.54}$$

Intuitively, the mutual information quantifies to what degree a single setting of the parameters agrees with the predictive posterior that marginalises over the parameters. If the predictions under all plausible parameter settings of the posterior are the same – whether they are all predicting the same class with probability 1 or making uniform predictions, i.e. low or high risk – the two entropies in Eq. 2.54 are identical and cancel out. The mutual information being 0 indicates that there is no model uncertainty. Conversely, if all predictions are made with probability 1, but uniformly distributed over the class labels, the marginal predictive posterior is a uniform distribution and has entropy $-\log C$, while each conditional predictive distribution has 0 entropy and the mutual information is high, indicating high model uncertainty.

Chapter 3

The Kronecker factored Laplace approximation

The Laplace approximation has a long history in the literature on BNNs. After first having been used in [55] and [38], with a diagonal approximation to the Hessian in the former, MacKay [220, 222] developed a unifying framework around the Laplace approximation that estimates predictive uncertainty, sets hyperparameters such as the prior precision and data noise for regression, and prunes network parameters in a principled way based on the 'evidence framework' [104]. In particular, MacKay strongly argues for the use of the full Hessian rather than a cheap diagonal approximation in order to account for parameter correlations in the posterior. See also [73] for a recent discussion of the importance of accounting for parameter correlations in BNNs. Further, [47] observes that with a fully factorised approximate posterior in the infinite width-limit of a single-layer network the mean function becomes 0, i.e. the BNN ignores the data.

While most recent works on Bayesian deep learning have focused on developing variational inference and MCMC approaches to match the mass of the posterior as closely as possible, the Laplace approximation holds the unique advantage of being applicable to already trained networks. This make it a highly practical approach for settings where a need for accurate uncertainty estimation may only emerge post-training.

Using the full Hessian is generally infeasible for modern network architectures

due to their number of parameters typically being in the millions to billions, so approximations are needed. This chapter develops a Laplace approximation to the neural network posterior that can scale to modern-sized architectures while taking per-layer parameter correlations into account. Section 3.1 introduces the curvature approximations used in this work and the corresponding approximate posterior distribution based on material from [29] and [289], with an added discussion on recent work analysing the loss landscape of neural networks in Section 3.1.4. Section 3.2 presents the details regarding approximating the predictive posterior using samples from the approximate distribution over the weights, alongside the experimental results in [289], with an added discussion of the impact of the curvature approximations on the concentration of the resulting posterior in Section 3.2.4. Section 3.3 extends the approximation to continual learning settings based on [288].

Statement of contributions The work in this chapter was done in collaboration with my colleague Alex Botev and my supervisor David Barber. Alex and David developed the approximation to the Gauss-Newton of neural networks published in [29], Alex wrote most of the code and I assisted with some experiments. Alex and I then jointly developed a Theano [328] and Lasagne [58] implementation for the approximate curvature matrices as the foundation for the sampling code of [289] and the quadratic penalties in [288], for which the three of us had developed the theoretical framework together. I designed and implemented the experiments for both papers with feedback from Alex and David; Alex and I wrote the papers together with feedback from David and this thesis re-uses large sections of all three papers in a restructured and unified notation, which hopefully makes the present text easier to follow. Most of the content of this chapter is also presented in Alex's PhD thesis with additional material on 2nd-order optimisation [28] from [29], which has been omitted from the present thesis.

3.1 Approximations

Approximating the posterior of a neural network with the Laplace approximation comes with two core challenges: 1. the loss function, i.e. the unnormalised posterior,

is not convex, hence the Hessian is not guaranteed to be p.s.d. 2. the size of the Hessian scales quadratically in the number of parameters, which is typically in the millions, but can be in the billions for modern architectures [311, 277, 280]. It is therefore infeasible to work with the full Hessian, leading to a need for approximations.

This section discusses how the Hessian can be approximated in a computationally efficient manner and how these approximations impact the structure of corresponding approximate posterior distribution.

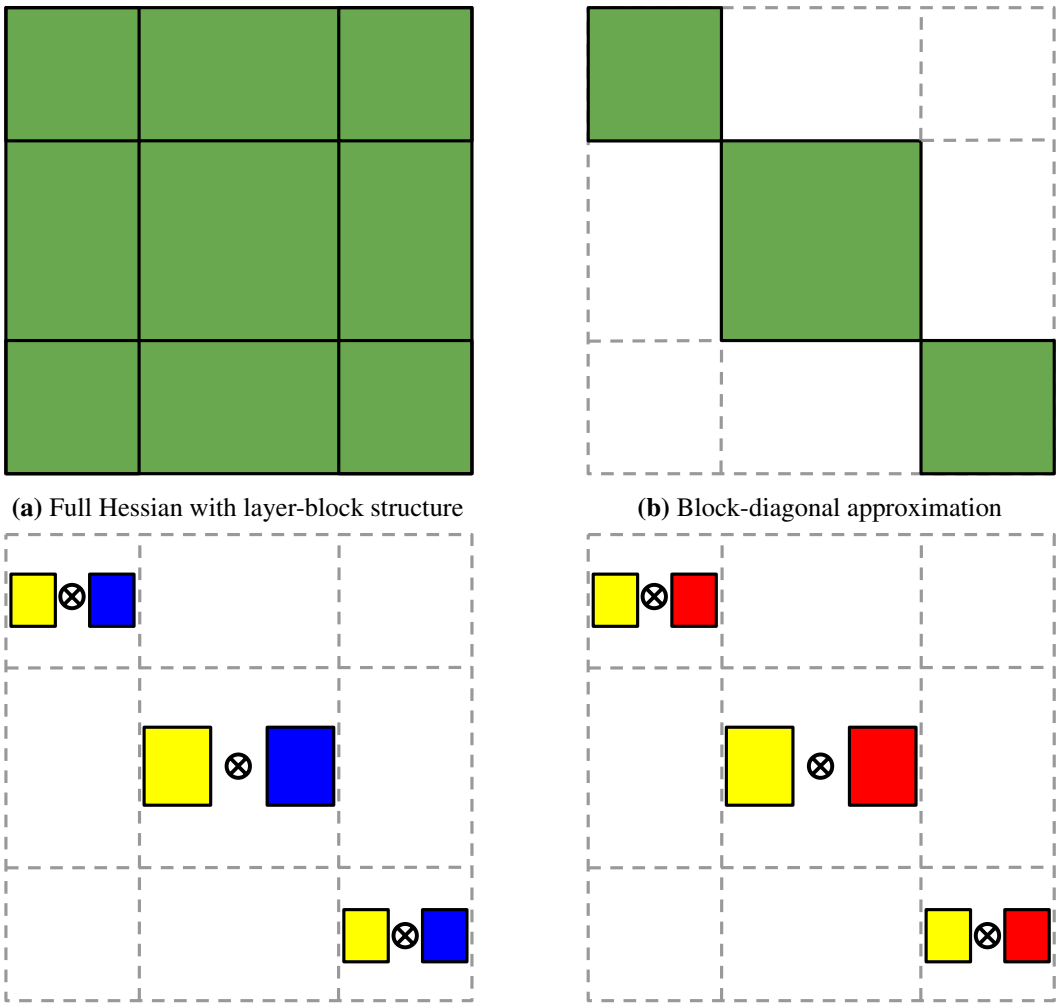
3.1.1 Curvature approximations

On a high level, we make three approximations to the Hessian: a block-diagonal one – treating each layer independently – followed by a Kronecker factored approximation of each layer's Hessian and finally a positive-definite approximation by the Fisher or Gauss-Newton. We visualise these approximations in Fig. 3.1 to give an intuition for the corresponding savings in the size of the matrix. Note that the block-diagonal and Kronecker factored approximation could also be developed directly for the Fisher or Gauss-Newton as e.g. in [229], which are positive-definite by construction, but we keep the discussion here in terms of the Hessian for generality.

3.1.1.1 Approximating the log likelihood Hessian

As stated above, working with the full Hessian is infeasible for all but the smallest network architectures. Even for a by current standards modestly sized ResNet-18 [114, 115] at 11.7 M parameters, storing all entries of the Hessian in single precision requires almost 500 TB (or half of that when taking advantage of the symmetry of the Hessian).

As a reminder, we denote a feedforward network as taking an input $\mathbf{a}_0 = \mathbf{x}$ and producing an output $\mathbf{h}_L$. The intermediate representations for layers $l = 1, ..., L$ are denoted as $\mathbf{h}_l = \boldsymbol{W}_l \mathbf{a}_{l-1}$ and $\mathbf{a}_l = f_l(\mathbf{h}_l)$. We refer to $\mathbf{a}_l$ as the activations, and $\mathbf{h}_l$ as the (linear) pre-activations. The bias terms are absorbed into the $\boldsymbol{W}_l$ by appending a 1 to each $\mathbf{a}_l$. The network parameters are optimised w.r.t. an error function $E(\mathbf{y}, \mathbf{h}_L)$ for targets $\mathbf{y}$ plus a a regularisation term that is function of the parameters. Most



(a) Full Hessian with layer-block structure

(b) Block-diagonal approximation

(c) KF approximation of the blocks

(d) Positive semi-definite approximation

Figure 3.1: Visualisation of the sequence of Hessian approximations.

commonly used error functions, such as squared error and categorical cross-entropy, can be interpreted as exponential family negative log likelihoods $-\log p(y|\mathbf{h}_L)$. The regulariser is usually equivalent to the negative log probability of prior for the parameters and typically has a simple second derivative, e.g. the identity matrix times the precision for an isotropic Gaussian, so we will focus the discussion on the Hessian of the negative log likelihood.

We will group the parameters by layer, i.e. $\boldsymbol{\theta} = [\boldsymbol{\theta}_1^\top, \ldots, \boldsymbol{\theta}_L^\top]^\top = [\mathrm{vec}(\boldsymbol{W}_1)^\top, \ldots, \mathrm{vec}(\boldsymbol{W}_L)^\top]$, where vec stacks the rows of a matrix into a single vector. In the following, all vectors and matrices, indicated by bold font, will have layer indices l as subscripts, scalar elements of these will carry them as superscripts

with the subscripts indicating the index. So $\boldsymbol{W}_l$ is the weight matrix of layer l and $W^l_{i,j}$ its entry in row i, column j.

The Hessian *for a single data point* is formally defined as

$$\boldsymbol{H} = \frac{\partial^2 E}{\partial \boldsymbol{\theta} \partial \boldsymbol{\theta}} = \begin{pmatrix} \boldsymbol{H}_{11} & \dots & \boldsymbol{H}_{1L} \\ \vdots & \ddots & \vdots \\ \boldsymbol{H}_{L1} & \dots & \boldsymbol{H}_{LL} \end{pmatrix} = \begin{pmatrix} \frac{\partial^2 E}{\partial \boldsymbol{\theta}_1 \partial \boldsymbol{\theta}_1} & \dots & \frac{\partial^2 E}{\partial \boldsymbol{\theta}_1 \partial \boldsymbol{\theta}_L} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial \boldsymbol{\theta}_L \partial \boldsymbol{\theta}_1} & \dots & \frac{\partial^2 E}{\partial \boldsymbol{\theta}_L \partial \boldsymbol{\theta}_L} \end{pmatrix}, \tag{3.1}$$

with the block structure following of the Hessian following the layer structure of the network, i.e. each block of the Hessian corresponds to the mixed partial Hessian w.r.t. to the parameters of two layers.

As a first approximation, we will treat the layers independently, i.e. approximate the Hessian as block-diagonal

$$\boldsymbol{H} \approx \begin{pmatrix} \boldsymbol{H}_1 & & 0 \\ & \ddots & \\ 0 & & \boldsymbol{H}_L \end{pmatrix} = \operatorname{diag}(\boldsymbol{H}_1, \dots, \boldsymbol{H}_L), \tag{3.2}$$

where we simplify the notation of the diagonal blocks to $\boldsymbol{H}_l = \boldsymbol{H}_{ll}$. Assuming we are dealing with a network where inputs, outputs and all layers are of dimensionality d, i.e. all $\boldsymbol{W}_l$ are of size $d \times d$, this block-diagonal approximation reduces the storage requirements from $\mathcal{O}(L^2 d^4)$ to $\mathcal{O}(L d^4)$. Further, it allows for most relevant linear algebra operations, such as inverses, the Cholesky decomposition and matrix-vector products to be computed independently for each block.

The size of the blocks is, however, still problematic. For example, the block corresponding to a 1024×1024 layer, a common building block for architectures on small datasets such as MNIST [194], would require $4TB$ to store. Hence, further approximations are needed.

Fortunately, the diagonal blocks of the Hessian have a structure that can be exploited for an efficient approximation. Starting with the gradient w.r.t. a specific

weight $W^l_{i,j}$, we apply the chain rule once

$$\frac{\partial E}{\partial W^l_{i,j}} = \sum_k \frac{\partial \mathrm{h}^l_k}{\partial W^l_{i,j}} \frac{\partial E}{\partial \mathrm{h}^l_k} = \mathrm{a}^{l-1}_j \frac{\partial E}{\partial \mathrm{h}^l_i}. \tag{3.3}$$

Differentiating again w.r.t. another weight $W^l_{m,n}$ from the same layer yields

$$\frac{\partial^2 E}{\partial W^l_{i,j} \partial W^l_{m,n}} = \mathrm{a}^{l-1}_j \sum_k \frac{\partial \mathrm{h}^l_k}{\partial W^l_{m,n}} \frac{\partial^2 E}{\partial \mathrm{h}^l_i \partial \mathrm{h}^l_k} = \mathrm{a}^{l-1}_j \mathrm{a}^{l-1}_n \frac{\partial^2 E}{\partial \mathrm{h}^l_i \partial \mathrm{h}^l_m}. \tag{3.4}$$

Noting the independent indices on the elements of the left and right factor, we can equivalently express this in matrix form as a Kronecker product

$$\boldsymbol{H}_l = \frac{\partial^2 E}{\partial \boldsymbol{\theta}_l \partial \boldsymbol{\theta}_l} = (\mathbf{a}_{l-1} \mathbf{a}^\top_{l-1}) \otimes \frac{\partial^2 E}{\partial \mathbf{h}_l \partial \mathbf{h}_l} =: \boldsymbol{\mathcal{Q}}_l \otimes \boldsymbol{\mathcal{H}}_l, \tag{3.5}$$

where $\mathbf{a}_{l-1} = [\mathrm{a}^{l-1}_1, \dots, \mathrm{a}^{l-1}_d]^\top$ and $\mathbf{h}_l = [\mathrm{h}^l_1, \dots, \mathrm{h}^l_d]^\top$. The Kronecker product $\otimes$ constructs a single large matrix by multiplying each element from the left matrix by the entire right matrix and tiling the results, s.t.

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{pmatrix} A_{11}\boldsymbol{B} & \cdots & A_{1n}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ A_{m1}\boldsymbol{B} & \cdots & A_{mn}\boldsymbol{B} \end{pmatrix} \tag{3.6}$$

for some matrices $\boldsymbol{A} : m \times n$ and $\boldsymbol{B}$ of arbitrary shape. This allows for many operations to be computed efficiently as a function of the individual Kronecker factors, so that the resulting matrix never needs to be computed, see Tab. 3.1 for a summary. Still assuming all square weight matrices, the Kronecker factors will be of shape $d \times d$, i.e. reducing the overall storage cost from $\mathcal{O}(Ld^4)$ to $\mathcal{O}(Ld^2)$. For arbitrarily shaped weights $\boldsymbol{W} : d_{out} \times d_{in}$, we will have $\boldsymbol{\mathcal{Q}} : d_{in} \times d_{in}$ and $\boldsymbol{\mathcal{H}} : d_{out} \times d_{out}$.

Table 3.1: Kronecker product identities. Note that in the vector product identity the placement of the transpose depends on the definition of the vec operation. The identity here is shown for concatenating the rows (equivalent to flattening in 'C-order'), when stacking the columns (flattening in 'Fortran-order'), the $\boldsymbol{A}$ rather than the $\boldsymbol{B}$ matrix is transposed.

Operation	Raw expression	Efficient identity
Inverse	$(\boldsymbol{A}\otimes\boldsymbol{B})^{-1}$	$\boldsymbol{A}^{-1}\otimes\boldsymbol{B}^{-1}$
Cholesky	$\boldsymbol{L}\boldsymbol{L}^\top=\boldsymbol{A}\otimes\boldsymbol{B}$	$(\boldsymbol{L}_A\otimes\boldsymbol{L}_B)(\boldsymbol{L}_A^\top\otimes\boldsymbol{L}_B^\top)$ s.t. $\boldsymbol{L}_A\boldsymbol{L}_A^\top=\boldsymbol{A},\boldsymbol{L}_B\boldsymbol{L}_B^\top=\boldsymbol{B}$
Vector product	$(\boldsymbol{A}\otimes\boldsymbol{B})\,\mathrm{vec}(\boldsymbol{X})$	$\mathrm{vec}(\boldsymbol{B}^\top\boldsymbol{X}\boldsymbol{A})$

Similar to the gradients, $\boldsymbol{\mathcal{H}}_l$, the Hessian w.r.t. the linear pre-activations of a layer, can be calculated recursively as

$$\boldsymbol{\mathcal{H}}_l=\boldsymbol{B}_l\boldsymbol{W}_{l+1}^\top\boldsymbol{H}_{l+1}\boldsymbol{W}_{l+1}\boldsymbol{B}_l+\boldsymbol{D}_l. \tag{3.7}$$

The diagonal matrices $\boldsymbol{B}_l$ and $\boldsymbol{D}_l$ are defined as

$$\boldsymbol{B}_l=\mathrm{diag}(f_l'(\mathbf{h}_l)) \tag{3.8}$$

$$\boldsymbol{D}_l=\mathrm{diag}(f_l''(\mathbf{h}_l)\frac{\partial E}{\partial \mathbf{a}_l}), \tag{3.9}$$

where f' and f'' denote the first and second derivative of the elementwise non-linearity respectively, and the recursion is initialised with the Hessian w.r.t. the network outputs, i.e.$\boldsymbol{\mathcal{H}}_L=\frac{\partial^2 E}{\partial \mathbf{h}_L{}^2}$. See Appendix A for the detailed derivation.

Typically, we will need the *expected* Hessian over some dataset rather than a single datapoint. Making the dependence on the data explicit, the Hessian is

$$\mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[\boldsymbol{H}\right]\approx\frac{1}{N}\sum_{\mathbf{x},\mathbf{y}\in\mathcal{D}}\frac{\partial^2 E(x,y)}{\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}}. \tag{3.10}$$

Unfortunately, Kronecker products do not add up pairwise i.e. $\boldsymbol{A}\otimes\boldsymbol{B}+\boldsymbol{C}\otimes\boldsymbol{D}\neq(\boldsymbol{A}+\boldsymbol{C})\otimes(\boldsymbol{B}+\boldsymbol{D})$ for arbitrary compatible matrices. Therefore, the Hessian loses its Kronecker product structure in expectation.

To maintain this computationally efficient form, we can approximate the blocks as being statistically independent

$$\mathbb{E}[\boldsymbol{H}_l] \approx \mathbb{E}[\boldsymbol{\mathcal{Q}}_l] \otimes \mathbb{E}[\boldsymbol{\mathcal{H}}_l]. \tag{3.11}$$

3.1.1.2 Positive definite approximations

Finally, we need to ensure that our curvature matrix in Eq. 3.11 is positive definite, which for a Kronecker product is the case if both factors are. While $\boldsymbol{\mathcal{Q}}$ is p.s.d. by construction, $\boldsymbol{\mathcal{H}}$ may not be due to the non-linear nature of neural networks, so a further approximation is needed.

Gauss-Newton One option is the Gauss-Newton matrix as presented in [29]. In general, we can decompose the Hessian into a sum of two matrices by applying the chain rule once starting from the outputs as

$$\boldsymbol{H} = \frac{\partial^2 E}{\partial \boldsymbol{\theta}^2} = \underbrace{\frac{\partial \mathbf{h}_L^\top}{\partial \boldsymbol{\theta}} \frac{\partial^2 E}{\partial \mathbf{h}_L{}^2} \frac{\partial \mathbf{h}_L}{\partial \boldsymbol{\theta}}}_{\text{Gauss-Newton}} + \sum_d \frac{\partial E}{\partial \mathrm{h}_d^L} \frac{\partial^2 \mathbf{h}_L}{\partial \boldsymbol{\theta}^2}, \tag{3.12}$$

where the first summand is known as the 'Gauss-Newton'. If the Hessian of the error w.r.t. the network outputs is p.s.d., so is the Gauss-Newton matrix and we can use it as an approximation to the Hessian by dropping the second term. This is justified in particular near local minima, where the gradients of the error w.r.t. the network outputs are near 0. We can plug this definition into the Kronecker factored approximation of the Hessian and approximate the Hessian w.r.t. the layerwise pre-activations as

$$\boldsymbol{H}_l \approx \boldsymbol{\mathcal{Q}}_l \otimes \boldsymbol{\mathcal{G}}_l = \boldsymbol{\mathcal{Q}}_l \otimes \frac{\partial \mathbf{h}_L^\top}{\partial \mathbf{h}_l} \boldsymbol{H}_L \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_l}. \tag{3.13}$$

The Gauss-Newton matrix had first been used in MacKay's Laplace framework in [75]. It can be calculated using Eq. 3.7 by dropping the $\boldsymbol{D}_l$ term. Interestingly, if the non-linear functions are piecewise linear, i.e. have zero second derivatives, the $\boldsymbol{D}_l$ matrices become zero and the Hessian blocks are equal to the blocks of the Gauss-Newton. Hence using the Gauss-Newton is not necessarily an additional

approximation on top of the block-diagonal one, e.g. for networks with ReLU non-linearities as discussed in [29] in more detail.

Efficient computation A naive per-datapoint implementation fails to fully utilise modern parallelised hardware and batching across many data quickly becomes prohibitive in terms of memory. If the dimensionality of the output layer is relatively small, e.g. in classification, we can calculate the square root of $\boldsymbol{H}_L : d_L \times d_L$

$$\boldsymbol{H}_L = \boldsymbol{C}_L \boldsymbol{C}_L^\top, \tag{3.14}$$

and then compute the per-layer square roots of the Gauss-Newton as

$$\boldsymbol{C}_l = \boldsymbol{B}_l \boldsymbol{W}_{l+1}^\top \boldsymbol{C}_{l+1}, \tag{3.15}$$

where each $\boldsymbol{C}_l$ will be of shape $m \times d \times d_L$ for a batch of m data points, which will typically be acceptable for small d_L.

Alternatively, we can recursively backpropagate the average Gauss-Newton for an efficient approximation as presented in [29] as KFRA (Kronecker factored resursive approximation)

$$\mathbb{E}\left[\boldsymbol{G}\right]_l = \mathbb{E}\left[\boldsymbol{B}_l \boldsymbol{W}_{l+1}^\top \boldsymbol{G}_{l+1} \boldsymbol{W}_{l+1} \boldsymbol{B}_l\right] \approx \mathbb{E}\left[\boldsymbol{B}_l \boldsymbol{W}_{l+1}^\top \mathbb{E}\left[\boldsymbol{G}_{l+1}\right] \boldsymbol{W}_{l+1} \boldsymbol{B}_l\right]. \tag{3.16}$$

Our Theano [328] implementation for Lasagne [58] is available at https://github.com/BB-UCL/Lasagne. Implementations for PyTorch [269] have been independently made available in the more recent Backpack library [51] after completion of this work.

Fisher information As an alternative to the Gauss-Newton, we can use the Fisher information matrix, which had originally been developed in frequentist statistics to measure the sampling statistics of maximum likelihood estimators [71, 72]. Its use has been popularised as natural gradient descent [6, 267, 7], corresponding to locally steepest descent in distribution rather than parameter space. The Fisher for some

fixed values $\boldsymbol{\theta}^*$ of the parameters is defined as

$$\boldsymbol{F} = \mathbb{E}_{p(\mathbf{x})p(y|\boldsymbol{\theta}^*,\mathbf{x})}\left[\nabla_{\boldsymbol{\theta}} \log p\left(y|\boldsymbol{\theta}^*,\mathbf{x}\right) \nabla_{\boldsymbol{\theta}} \log p\left(y|\boldsymbol{\theta}^*,\mathbf{x}\right)^\top\right] \tag{3.17}$$

$$= -\mathbb{E}\left[\nabla^2_{\boldsymbol{\theta}} \log p\left(y|\boldsymbol{\theta}^*,\mathbf{x}\right)\right] \tag{3.18}$$

$$= \mathbb{E}\left[\nabla^2_{\boldsymbol{\theta}} E(\boldsymbol{\theta}^*)\right] = \mathbb{E}\left[\boldsymbol{H}\right]. \tag{3.19}$$

Note that the expectation is over the predictive distribution of the network, not the empirical data distribution. The latter has been used as an approximation in various pieces of recent work in deep learning as the 'empirical Fisher', however this matrix has been shown to not reliably capture curvature information [183].

We can then approximate each block of the Hessian in Eq. 3.11 as the corresponding block of the Fisher as

$$\boldsymbol{H}_l \approx \boldsymbol{\mathcal{Q}}_l \otimes \boldsymbol{\mathcal{F}}_l = \boldsymbol{\mathcal{Q}}_l \otimes \mathbb{E}\left[\nabla_{\mathbf{h}_l} E(\boldsymbol{\theta}^*) \nabla_{\mathbf{h}_l} E(\boldsymbol{\theta}^*)^\top\right]. \tag{3.20}$$

This approximation has first been discussed in [125] and more recently developed into a practical second-order optimisation method for neural networks under the name KFAC (Kronecker factored approximate curvature) [229, 103, 274, 261, 262].

For exponential-family likelihoods, the Fisher and Gauss-Newton are equivalent [228]. We have shown above that, with piecewise-linear non-linearities, the layerwise blocks of the Hessian and Gauss-Newton are identical and the block-diagonal Hessian is therefore p.s.d. for many architectures. Therefore, we will keep all discussion in the following in terms of the Hessian for full generality and regard the final positive definite approximation as an optional final step. Implementationwise, it is typically easiest to work with the Fisher as it only requires calculating the Jacobian of the loss over a batch of data w.r.t. to the parameters. While efficient implementations are not available for most automatic differentiation packages out-of-the-box, workarounds for the commonly used network layer types can be coded at a manageable overhead [94, 51].

Convolutional layers Besides fully connected layers, we also need to consider convolutional layers, which have been instrumental in the success of neural networks

for computer vision [54]. We can exploit the connection between convolutional and linear layers as described in Section 2.1.1.2 of a convolution being a linear operation applied to a batch of image patches. Assuming independence across these locations, similar to the independence assumption between data, then gives a corresponding Kronecker product approximation of the Hessian as also discussed in [103].

3.1.2 Posterior approximations

We will now interpret the curvature approximations from a posterior-approximation point of view. For the standard Laplace approximation, we use the full Hessian (assuming it is positive definite) of the negative log likelihood plus the Hessian of the negative log prior as the precision, i.e. the inverse covariance. Hence we need N times the average Hessian

$$\bar{\boldsymbol{H}} = N\,\mathbb{E}\left[\boldsymbol{H}\right], \tag{3.21}$$

and add the negative log prior Hessian. Assuming an isotropic Gaussian prior make the Hessian of the negative log posterior around the MAP estimate

$$\tilde{\boldsymbol{H}} = \bar{\boldsymbol{H}} + \tau\boldsymbol{I}, \tag{3.22}$$

so that the overall posterior is approximated as

$$p\left(\boldsymbol{\theta}\,|\,\mathcal{D}\right) \approx q_{Full}(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}^*, \tilde{\boldsymbol{H}}^{-1}). \tag{3.23}$$

With the block-diagonal approximation, i.e.

$$\boldsymbol{H} \approx \operatorname{diag}(\boldsymbol{H}_1, \ldots, \boldsymbol{H}_L), \tag{3.24}$$

the approximate posterior factorises across the layers

$$p\left(\boldsymbol{\theta}\,|\,\mathcal{D}\right) \approx q_{BD}(\boldsymbol{\theta}) = \prod_{l=1}^{L} \mathcal{N}(\theta_l^*, \tilde{\boldsymbol{H}}_l^{-1}). \tag{3.25}$$

For the Kronecker factored approximation we define

$$\bar{\boldsymbol{\mathcal{Q}}}_l = \sqrt{N}\,\mathbb{E}\left[\boldsymbol{\mathcal{Q}}_l\right] \quad \text{and} \quad \tilde{\boldsymbol{\mathcal{Q}}}_l = \bar{\boldsymbol{\mathcal{Q}}}_l + \sqrt{\tau}\boldsymbol{I}, \tag{3.26}$$

and

$$\bar{\boldsymbol{\mathcal{H}}}_l = \sqrt{N}\,\mathbb{E}\left[\boldsymbol{\mathcal{H}}_l\right] \quad \text{and} \quad \tilde{\boldsymbol{\mathcal{H}}}_l = \bar{\boldsymbol{\mathcal{H}}}_l + \sqrt{\tau}\boldsymbol{I}, \tag{3.27}$$

so that overall, the block of the negative log posterior for each layer is approximated as

$$\tilde{\boldsymbol{H}}_l \approx \tilde{\boldsymbol{\mathcal{Q}}}_l \otimes \tilde{\boldsymbol{\mathcal{H}}}_l. \tag{3.28}$$

The precision and therefore the covariance of the approximate Gaussian posterior is hence Kronecker factored.

A multivariate normal distribution with Kronecker factored covariance is also known as matrix normal distribution [107], as the covariance factorises into a row and column covariance. In conjunction with the block-diagonal approximation, our approximate posterior is then

$$p\left(\boldsymbol{\theta} \,|\, \mathcal{D}\right) \approx q_{KF}(\boldsymbol{\theta}) = \prod_{l=1}^{L} \mathcal{MN}(\boldsymbol{W}_l^*, \tilde{\boldsymbol{\mathcal{H}}}_l^{-1}, \tilde{\boldsymbol{\mathcal{Q}}}_l^{-1}), \tag{3.29}$$

where $\tilde{\boldsymbol{\mathcal{H}}}_l^{-1}$ is the row and $\tilde{\boldsymbol{\mathcal{Q}}}_l^{-1}$ the column covariance.

3.1.3 Diagonal Approximation

For a computationally more efficient baseline that does not account for parameter correlations, we will also use a diagonal approximation. An approximation that is easy to compute in modern automatic differentiation frameworks is the diagonal of

the Fisher matrix F, which is simply the expectation of the squared gradients

$$\tilde{\boldsymbol{H}} \approx \operatorname{diag}(\tilde{\boldsymbol{F}}) \tag{3.30}$$

$$= \operatorname{diag}\left(N\mathbb{E}\left[\nabla_{\boldsymbol{\theta}} \log p\left(\mathcal{D} \mid \boldsymbol{\theta}\right) \nabla_{\boldsymbol{\theta}} \log p\left(\mathcal{D} \mid \boldsymbol{\theta}\right)^{\top}\right] - \frac{\partial^2 \log p(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2}\right) \tag{3.31}$$

$$= \operatorname{diag}\left(N\mathbb{E}\left[\left(\nabla_{\boldsymbol{\theta}} \log p\left(\mathcal{D} \mid \boldsymbol{\theta}\right)\right)^2\right]\right) - \operatorname{diag}\left(\frac{\partial^2 \log p(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2}\right), \tag{3.32}$$

where diag extracts the diagonal of a matrix or turns a vector into a diagonal matrix. Such diagonal approximations to the curvature of a neural network have been used successfully for pruning the weights [192] and, more recently, for continual learning [170].

The diagonal approximation corresponds to modelling the weights with a Normal distribution with diagonal covariance

$$\boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{\theta}^*, \operatorname{diag}(\tilde{\boldsymbol{F}})^{-1}). \tag{3.33}$$

3.1.4 The loss landscape of neural networks

Developing a better understanding of the loss landscape of neural networks has been an active area of research in recent years. While most works aim to develop a better understanding of the optimisation dynamics and generalisation behaviour of deterministic networks, the equivalence of the commonly used neural network losses with the negative log likelihood in the posterior makes these results highly relevant for Bayesian deep learning in general and the Laplace approximation in particular.

Most important for the Laplace approximation, multiple small-scale empirical studies [297, 298] first observed that on classification problems with C classes the spectrum of eigenvalues of the Hessian consists of C outliers and a bulk of eigenvalues near 0. This means that the uncertainty of the Laplace approximation is strongly constrained in C directions by the Hessian of the negative log likelihood and depends on the curvature of the prior in all others. These results have been confirmed on larger networks and datasets by estimation of the density of the eigenspectrum [89] via Hessian-vector products, which can be calculated efficiently without instantiating

the Hessian [270]. Arjevani and Field [10] provide some theoretically-grounded support for these observations based on shallow ReLU networks. Ghorbani et al. [89] further note that the magnitude of the outliers can be reduced via batch normalisation, although Yao et al. [365] find the opposite to be the case for shallow architectures. Papyan [265] attribute these outliers to the Gauss-Newton part of the Hessian, inspiring confidence that the p.s.d. approximations capture the most important information. Further, Wu et al. [362] find that the eigenspaces of the Kronecker factored Hessian agrees well with that of the full blocks, supporting the independence assumption in Eq. 3.11. Papyan [266] however find that the distribution of eigenvalues can disagree and propose a class-based (computationally more expensive) correction.

Some results that draw the use of the quadratic approximation in question have been published in recent years. Besides it being a local, uni-modal approximation and the success of ensembles [185] demonstrating that the loss landscape is multi-modal beyond functionally identical symmetries in the parameter space, it has been observed that the modes of the loss are connected via almost constant trajectories [60, 85]. That means that the bottom of the loss landscape is more similar to a valley-like shape rather than isolated quadratic 'bowls' or islands as commonly used in one-dimensional visualisations. Frye et al. [78] further warn of gradient-flat regions, where the quadratic approximation becomes untrustworthy, but that the layerwise Kronecker-factored approximation may overcome this issue and therefore be not only computationally needed, but further act as a regulariser by increasing the concentration of the approximate posterior in some settings as will be discussed in Section 3.2.4. Asides from the flatness, He et al. [112] find that minima of deep modern architectures exhibit significant asymmetry and attribute this to batch normalisation. Finally, visualisations of the loss suggest that in particular skip connections make the loss more regular and that higher-order effects seem to be present particularly in wide minima [203].

Overall, these results do not paint a fully clear picture due to the complexity of neural networks arising from their non-linear nature. While there are some reasons for concern that our approximate posterior may not be a good fit even locally, the

irregularities in the loss surface affect any Gaussian approximation. And while we certainly cannot hope to fully represent the posterior, it is not unreasonable to expect that even with some degree of mismatch our Gaussian approximation will empirically perform better than standard point estimates of the parameters. So we now turn to developing methods for uncertainty estimation and continual learning based on these Laplace posteriors.

3.2 Uncertainty estimation

We will begin utilising and evaluating the posterior approximation we developed in the previous section by approximating the predictive posterior. We do so by approximating the integral with a Monte Carlo approximation, for which we need to draw multiple samples from our approximate posterior, which consists of independent matrix normal distributions as discussed previously.

3.2.1 Sampling

The matrix normal distribution [107] is a multivariate distribution over an entire matrix of shape $n \times p$ rather than just a vector. In contrast to the multivariate normal distribution, it is parameterised by two p.s.d. covariance matrices, $\boldsymbol{U} : n \times n$ and $\boldsymbol{V} : p \times p$, which indicate the covariance of the rows and columns respectively. In addition it has a mean matrix $\boldsymbol{M} : n \times p$.

For a regular multivariate normal distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$), we can draw a random sample $\mathbf{x}$ as

$$\mathbf{x} = \boldsymbol{\mu} + \boldsymbol{L}\boldsymbol{\varepsilon} \tag{3.34}$$

$$\text{with} \quad \boldsymbol{\Sigma} = \boldsymbol{L}_\Sigma \boldsymbol{L}_\Sigma^\top \quad \text{and} \quad \boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}), \tag{3.35}$$

assuming that we can draw a sample $\boldsymbol{\varepsilon}$ from the standard normal distribution, for which all modern numerical computation libraries provide functions, and with $\boldsymbol{L}_\Sigma$ being the lower Cholesky decomposition of $\boldsymbol{\Sigma}$. A vectorised sample from a matrix normal distribution $\boldsymbol{X} \sim \mathcal{MN}(\boldsymbol{M}, \boldsymbol{A}, \boldsymbol{B})$ corresponds to a sample from a normal distribution $\text{vec}(\boldsymbol{X}) \sim \mathcal{N}(\text{vec}(\boldsymbol{M}), \boldsymbol{B} \otimes \boldsymbol{A})$, where $\boldsymbol{A}$ is the row and $\boldsymbol{B}$ the column

covariance and the vectorisation is performed by stacking the rows of $\boldsymbol{X}$. However, samples can be drawn more efficiently as $\boldsymbol{X} = \boldsymbol{M} + \boldsymbol{L}_A \boldsymbol{E} \boldsymbol{U}_B$ with $\boldsymbol{E} \sim \mathcal{MN}(\boldsymbol{0}, \boldsymbol{I}, \boldsymbol{I})$, and $\boldsymbol{L}_A \boldsymbol{L}_A^\top = \boldsymbol{A}$ and $\boldsymbol{U}_B^\top \boldsymbol{U}_B = \boldsymbol{B}$. The sample $\boldsymbol{E}$ corresponds to a sample from a normal distribution of length np that has been reshaped to a $n \times p$ matrix. This is more efficient in the sense that we only need to calculate two matrix-matrix products of small matrices, rather than a matrix-vector product with one big one.

However, with the Laplace approximation we have only access to the precision matrix $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$, i.e. the inverse of the covariance. Numerically inverting the precision to compute the Cholesky may be numerically unstable [126]. However, for the upper Cholesky $\boldsymbol{U}_\Lambda$ of $\boldsymbol{\Lambda}$ we have

$$\boldsymbol{\Lambda} = \boldsymbol{U}_\Lambda^\top \boldsymbol{U}_\Lambda \;\Rightarrow\; \boldsymbol{\Sigma} = \boldsymbol{U}_\Lambda^{-1} \boldsymbol{U}_\Lambda^{-\top}. \tag{3.36}$$

Using any linear solver, we can therefore draw samples given only the precision matrix without numerically calculating an inverse

$$\mathbf{x} = \boldsymbol{\mu} + \boldsymbol{U}_\Lambda^{-1} \boldsymbol{\varepsilon}. \tag{3.37}$$

Similarly for the matrix normal, with $\boldsymbol{A}^{-1} = \boldsymbol{P} = \boldsymbol{U}_P \boldsymbol{U}_P^\top$ and $\boldsymbol{B}^{-1} = \boldsymbol{Q} = \boldsymbol{L}_Q \boldsymbol{L}_Q^\top$, we can draw a sample as

$$\boldsymbol{X} = \boldsymbol{M} + \boldsymbol{U}_P^{-1} \boldsymbol{E} \boldsymbol{L}_Q^{-1}, \tag{3.38}$$

without needing to calculate the inverses explicitly.

To summarise, to draw a sample from the Kronecker factored Laplace approximation, we compute for each layer

$$\boldsymbol{W} = \boldsymbol{W}^* + \boldsymbol{U}_{\mathcal{H}}^{-1} \boldsymbol{E} \boldsymbol{L}_{\mathcal{Q}}^{-1} \sim \mathcal{MN}(\boldsymbol{W}^*, \tilde{\mathcal{H}}^{-1}, \tilde{\mathcal{Q}}^{-1}), \tag{3.39}$$

with

$$\mathcal{H} = \boldsymbol{U}_{\mathcal{H}}^{\top} \boldsymbol{U}_{\mathcal{H}} \quad \text{and} \quad \mathcal{Q} = \boldsymbol{L}_{\mathcal{Q}} \boldsymbol{L}_{\mathcal{Q}}^{\top} \tag{3.40}$$

being the upper Cholesky of the activation covariance and the lower Cholesky of the pre-activation curvature respectively.

3.2.2 Regularising the posterior approximation

Unfortunately, sampling from the posterior approximation as introduced above may lead to poor predictive performance, either due to a mis-match between the Laplace approximation and the true posterior or issues with the true posterior itself, see e.g. [354]. Hence it can be desirable to adjust the spread of the mass of the Gaussian posterior around its mean. To achieve this, recall that just as the log posterior, the Hessian decomposes into a term depending on the data log likelihood and one on the prior. For the commonly used L_2-regularisation, corresponding to a Gaussian prior, the Hessian is equal to the precision of the prior times the identity matrix. We approximate this by adding a multiple of the identity to each of the Kronecker factors from the log likelihood

$$\tilde{\boldsymbol{H}}_l = N \mathbb{E}\left[-\frac{\partial^2 \log p(\mathcal{D}|\boldsymbol{\theta})}{\partial \theta^2} \right] + \tau \boldsymbol{I} \approx (\sqrt{N}\mathbb{E}[\boldsymbol{\mathcal{Q}}_l] + \sqrt{\tau}\boldsymbol{I}) \otimes (\sqrt{N}\mathbb{E}[\boldsymbol{\mathcal{H}}_l] + \sqrt{\tau}\boldsymbol{I}), \tag{3.41}$$

where τ is the precision of the Gaussian prior on the weights and N the size of the dataset. However, we can also treat them as hyperparameters and optimise them w.r.t. the predictive performance on a validation set. We emphasise that this can be done without retraining the network, so it does not impose a large computational overhead and is trivial to parallelise.

Setting N to a larger value than the size of the dataset can be interpreted as

including duplicates of the data points as pseudo-observations. Adding a multiple of the uncertainty to the precision matrix decreases the uncertainty about each parameter. This has a regularising effect both on our approximation to the true Laplace, which may be overestimating the variance in certain directions due to ignoring the covariances between the layers, as well as the Laplace approximation itself, which may be placing probability mass in low probability areas of the true posterior. See also [145] for a visualisation.

3.2.3 Memory and Computational Requirements

If we denote the dimensionality of the input to layer l as D_{l-1} and its output as D_l, the curvature factors correspond to the two precision matrices with $\frac{D_{l-1}(D_{l-1}+1)}{2}$ and $\frac{D_l(D_l+1)}{2}$ 'parameters' to estimate, since they are symmetric. So across a network, the number of curvature directions that we are estimating grows linearly in the number of layers and quadratically in the dimension of the layers, i.e. the number of columns of the weight matrices. The size of the full Hessian, on the other hand, grows quadratically in the number of layers and with the fourth power in the dimensionality of the layers (assuming they are all the same size).

Once the curvature factors are calculated, which only needs to be done once, we use their Cholesky decomposition to solve two triangular linear systems when sampling weights from the matrix normal distribution. We use the same weight samples for each minibatch, i.e. we do not sample a weight matrix per datapoint. This is for computational efficiency and does not change the expectation.

One possibility to save computation time would be to sample a fixed set of weight matrices from the approximate posterior — in order to avoid solving the linear system on every forward pass — and treat the networks that they define as an ensemble. The individual ensemble members can be evaluated in parallel and their outputs averaged, which can be done with a small overhead over evaluating a single network given sufficient compute resources. A further speed up can be achieved by distilling the predictive distributions of the Laplace network into a smaller, deterministic feedforward network as successfully demonstrated in [16] for posterior samples using HMC.

Finally, in contrast to second-order optimisation methods, we do not need to approximate $\mathbb{E}[\boldsymbol{\mathcal{H}}]$ as it is only calculated once. However, when it is possible to augment the data (e.g. randomised cropping of images), it may be advantageous. While the square root of $\boldsymbol{\mathcal{Q}}$ is calculated during the forward pass on all layers, $\boldsymbol{\mathcal{H}}$ requires an additional backward pass. Strictly speaking, it is not essential to approximate $\mathbb{E}[\boldsymbol{\mathcal{H}}]$ for the Kronecker factored Laplace approximation, as in contrast to optimisation procedures the curvature only needs to be calculated once and is thus not time critical. For datasets of the scale of ImageNet and the networks used for such datasets, it would still be impractically slow to perform the calculation for every data point individually. Furthermore, as most datasets are augmented during training, e.g. random cropping or reflections of images, the curvature of the network can be estimated using the same augmentations, effectively increasing the size of the dataset by orders of magnitude. Thus, we make use of the minibatch approximation in our experiments — as we make use of data augmentation — in order to demonstrate its practical applicability.

We note that $\mathbb{E}[\boldsymbol{\mathcal{H}}]$ can be calculated exactly by running KFRA [29] with a minibatch-size of one, and then averaging the results. KFAC [229], in contrast, stochastically approximates the Fisher matrix in typical implementations, so even when run for every datapoint separately, it does not calculate the curvature factor exactly unless the expectation over the predictive posterior is enumerated.

3.2.4 Posterior concentration

To gain some intuition for how the different curvature approximations impact the uncertainty of the posterior, we will now compare the corresponding entropies. As all Laplace-based approximate posteriors are Gaussians, the entropy is generally given as

$$\mathbb{H}(\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})) = \frac{1}{2} \log \det(2\pi \mathrm{e} \boldsymbol{\Sigma}), \tag{3.42}$$

i.e. it is determined by the determinant of the inverse (approximate) Hessian. We will initially assume that the prior adds no curvature, e.g. a Laplace or (improper) uniform prior. This simplifies the comparison of the determinants of the Hessian

approximations for the negative log likelihood. Beginning our discussion with the structural approximations, i.e. the block-diagonal and diagonal ones compared to the full Hessian, with Fischer's inequality we have that

$$\det(\bar{\boldsymbol{H}}) \leq \det(\bar{\boldsymbol{H}}_{BD}) \leq \det(\operatorname{diag}(\bar{\boldsymbol{H}})). \tag{3.43}$$

Therefore, assuming that we have enough data points for all Normal distributions to be well-defined, i.e. none of the Hessians being rank-deficient, the corresponding entropies are sorted in the inverted order

$$\mathbb{H}(q_{Full}) \geq \mathbb{H}(q_{BD}) \geq \mathbb{H}(q_{diag}), \tag{3.44}$$

as the Hessians correspond to the precision matrices. Hence the diagonal approximation is more concentrated around the mean than the block-diagonal approximation, which in turn is more concentrated than the Laplace approximation corresponding to the full Hessian.

With the more commonly used Gaussian priors, however, we now have a multiple of the identity that is added to each Hessian. Even though

$$\det(\boldsymbol{A}+\boldsymbol{B}) \geq \det\boldsymbol{A} + \det\boldsymbol{B} \geq \det\boldsymbol{A}, \tag{3.45}$$

for $\boldsymbol{A}, \boldsymbol{B}$ p.s.d. i.e. the addition of curvature from the prior decreasing the posterior uncertainty, in general

$$\det\boldsymbol{A} \geq \det\boldsymbol{B} \nRightarrow \det(\boldsymbol{A}+\boldsymbol{I}) \geq \det(\boldsymbol{B}+\boldsymbol{I}), \tag{3.46}$$

e.g. with $\boldsymbol{A} = \begin{pmatrix} 5 & 0 \\ 0 & 6 \end{pmatrix}$ and $\boldsymbol{B} = \begin{pmatrix} 2 & 0 \\ 0 & 14 \end{pmatrix}$ we have $\det\boldsymbol{A} = 30 > 28 = \det\boldsymbol{B}$, but $\det(\boldsymbol{A}+\boldsymbol{I}) = 42 < 45 = \det(\boldsymbol{B}+\boldsymbol{I})$. Hence, with a Gaussian prior the block-diagonal and diagonal Laplace approximation do not necessarily concentrate their mass more tightly around the mode than the full Laplace approximation.

Similarly, as Kronecker products factorise the same way as scalar products, the

Kronecker factored approximation implies that each block of the average Hessian is approximated as

$$\mathbb{E}[\mathcal{Q}] \otimes \mathbb{E}[\mathcal{H}] = \left(\frac{1}{N}\sum_{i=1}^{N} \mathcal{Q}_i\right) \otimes \left(\frac{1}{N}\sum_{i=1}^{N} \mathcal{H}_i\right) \approx \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} \mathcal{Q}_i \otimes \mathcal{H}_j, \tag{3.47}$$

i.e. each block is now a sum over N^2 instead of N terms, which is accounted for by a division by N^2. So as the division correctly accounts for the additional (positive definite) terms compared to the exact block, the Kronecker factored approximation does not analytically lead to an increase or decrease of the posterior concentration.

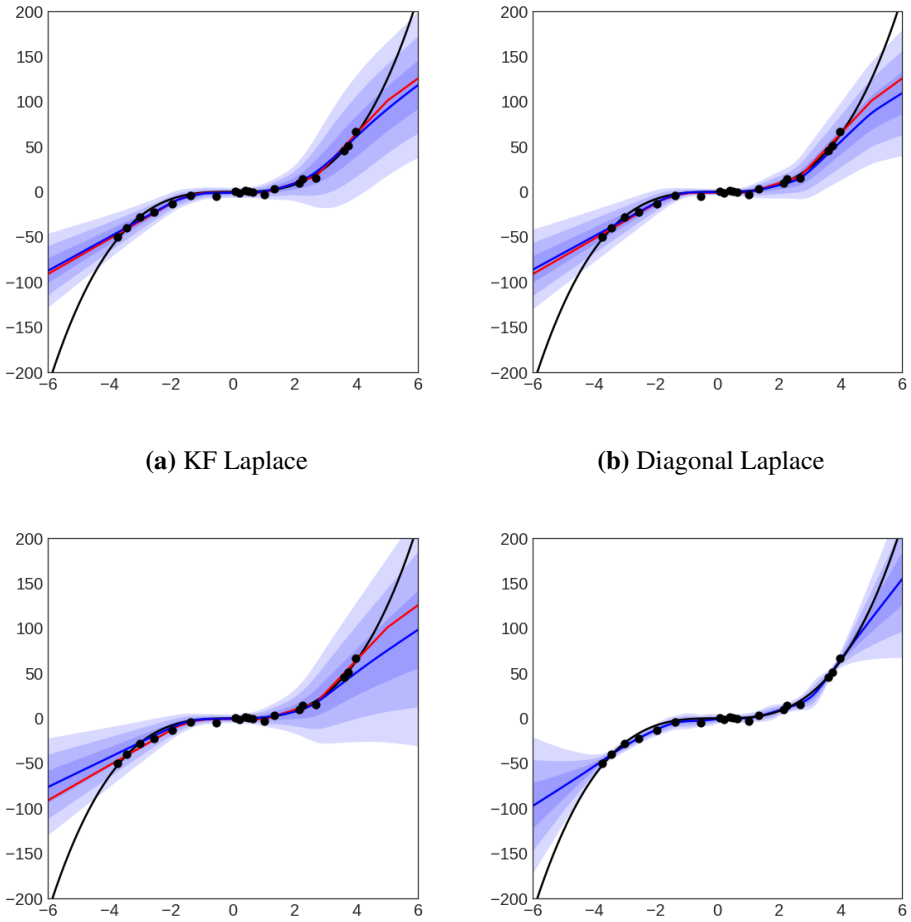
3.2.5 Experiments

Since the Laplace approximation is a method for *predicting* in a Bayesian manner and not for training, we focus on comparing to uncertainty estimates obtained from Dropout [83]. The trained networks will be identical, but the prediction methods will differ. We also compare to a diagonal Laplace approximation to highlight the benefit from modelling the covariances between the weights. All experiments are implemented using Theano [328] and Lasagne [58].

3.2.5.1 Toy regression dataset

As a first experiment, we visualise the uncertainty obtained from the Laplace approximations on a toy regression dataset, similar to [124]. We create a dataset of 20 uniformly distributed points $x \sim \mathcal{U}(-4,4)$ and sample $y \sim \mathcal{N}(x^3, 3^2)$. In contrast to [124], we use a two-layer network with seven units per layer rather than one layer with 100 units. This is because both the input and output are one-dimensional, hence the weight matrices are vectors and the matrix normal distribution reduces to a multivariate normal distribution. Furthermore, the Laplace approximation is sensitive to the ratio of the number of data points to parameters, and we want to visualise it both with and without hyperparameter tuning.

Fig. 3.2 shows the uncertainty obtained from the Kronecker factored and diagonal Laplace approximation applied to the same network, as well as from a full Laplace approximation and $50,000$ HMC [249] samples. The latter two methods are



(a) KF Laplace

(b) Diagonal Laplace

(c) Full Laplace

(d) HMC

Figure 3.2: Toy regression uncertainty. Black dots are data points, the black line shows the noiseless function. The red line shows the deterministic prediction of the network, the blue line the mean output. Each shade of blue visualises one additional standard deviation.

feasible only for such a small model and dataset. For the diagonal and full Laplace approximation we use the Fisher identity and draw one sample per data point. We set the hyperparameters of the Laplace approximations using a grid search over the likelihood of 20 validation points that are sampled the same way as the training set.

The regularised Laplace approximations give an overall good fit to the HMC predictive posterior. Their uncertainty is slightly higher close to the training data and increases more slowly away from the data than that of the HMC posterior. The diagonal and full Laplace approximation require stronger regularisation than our Kronecker factored one, as they have higher uncertainty when not regularised.

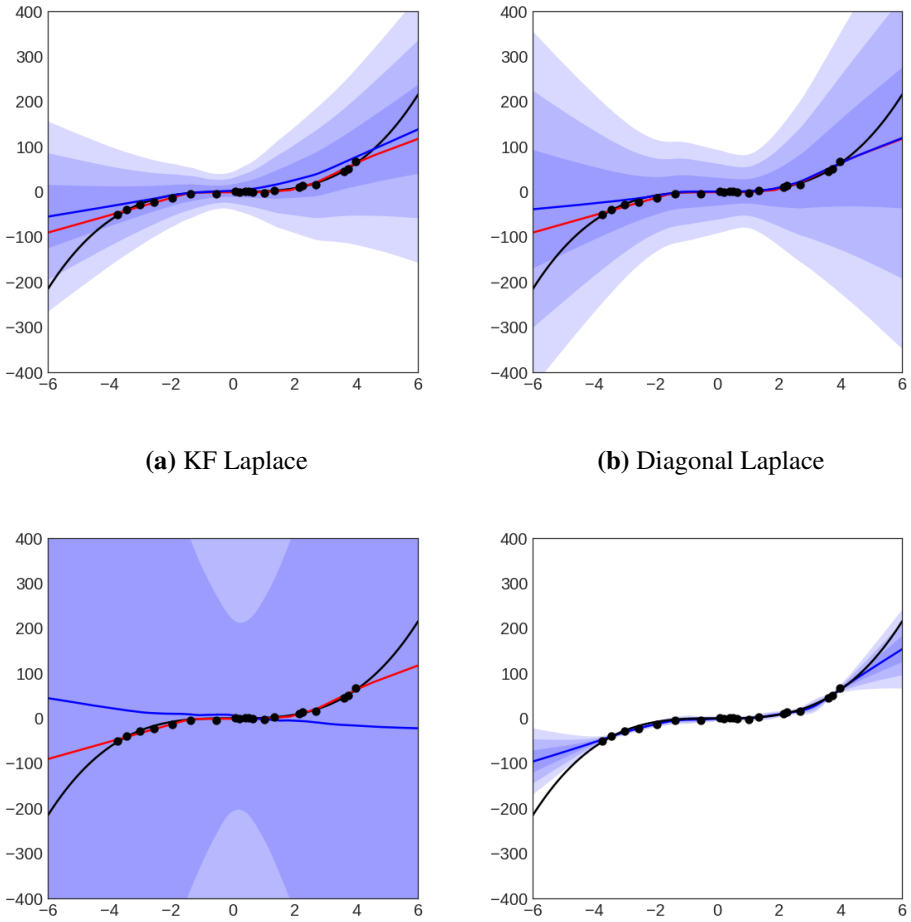
In particular the full Laplace approximation vastly overestimates the uncertainty without additional regularisation, leading to a bad predictive mean (see below), as the Hessian of the log likelihood is underdetermined. This is commonly the case in deep learning, as the number of parameters is typically much larger than the number of data points. Hence restricting the structure of the covariance is not only a computational necessity for most architectures, but also allows for more precise estimation of the approximate covariance.

Fig. 3.3 shows the different Laplace approximations (Kronecker factored, diagonal, full) without any hyperparameter tuning. The figure of the uncertainty obtained from samples using HMC is repeated. Note that the scale is larger than in Fig. 3.2 due to the high uncertainty of the Laplace approximations.

The Laplace approximations are increasingly uncertain away from the data, as the true posterior estimated from HMC samples, however they all overestimate the uncertainty without regularisation. This is easy to fix by optimising the hyperparameters on a validation set as discussed before, resulting in posterior uncertainty much more similar to the true posterior. As discussed in [29], the Hessian of a neural network is usually underdetermined as the number of data points is much smaller than the number of parameters — in our case we have 20 data points to estimate a 78×78 precision matrix. This leads to the full Laplace approximation vastly overestimating the uncertainty and a bad predictive mean, in line with results in [188]. Both the Kronecker factored and the diagonal approximation exhibit smaller variance than the full Laplace approximation as they restrict the structure of the precision matrix. Consistently with the other experiments, we find the diagonal Laplace approximation to place more mass in low probability areas of the posterior than the Kronecker factored approximation, resulting in higher variance on the regression problem. This leads to a need for greater regularisation of the diagonal approximation to obtain acceptable predictive performance, and underestimating the uncertainty.

3.2.5.2 Out-of-distribution uncertainty

For a more realistic test, similar to [215], we assess the uncertainty of the predictions when classifying data from a different distribution than the training data. For this we



(a) KF Laplace

(b) Diagonal Laplace

(c) Full Laplace

(d) HMC

Figure 3.3: Toy regression uncertainty. Black dots are data points, the black line shows the underlying noiseless function. The red line shows the deterministic prediction of the trained network, the blue line the mean output. Each shade of blue visualises one additional standard deviation.

Table 3.2: Test accuracy of the feedforward network trained on MNIST

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

train a network with two layers of 1024 hidden units and ReLU transfer functions to classify MNIST digits. We use a learning rate of 10^{-2} and momentum of 0.9 for 250 epochs. We apply Dropout with p=0.5 after each inner layer, as our chief interest is to compare against its uncertainty estimates. We further use L_2-regularisation with a factor of 10^{-2} and randomly binarise the images during training according

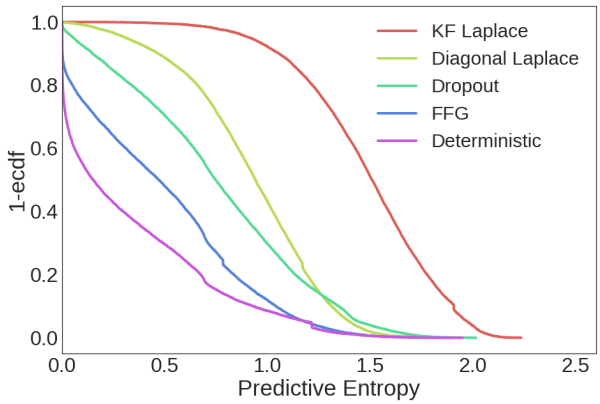


Figure 3.4: Empirical cumulative distribution of predictive entropy on notMNIST obtained from different methods for the forward pass on a network trained on MNIST. On OOD data, higher predictive entropy (with a maximum of $\log 10 \approx 2.3$ for a uniform distribution across the 10 classes in MNIST) is desirable, so an ideal predictor would pass through the top right corner and curves further up correspond to better performance.

to their pixel intensities and draw $1,000$ such samples per datapoint for estimating the curvature factors. We use this network to classify the images in the notMNIST dataset[1], which contains 28×28 grey-scale images of the letters 'A' to 'J' from various computer fonts, i.e. not digits. An ideal classifier would make uniform predictions over its classes.

We compare the uncertainty obtained by predicting the digit class of the notMNIST images using 1. a deterministic forward pass through the Dropout trained network, 2. by sampling different Dropout masks and averaging the predictions, and by sampling different weight matrices from 3. the matrix normal distribution obtained from our Kronecker factored Laplace approximation as well as 4. the diagonal one. As an additional baseline similar to [26, 100], we compare to a network with identical architecture with a fully factorised Gaussian (FFG) approximate posterior on the weights and a standard normal prior. We train the model on the variational lower bound using the reparameterisation trick [167]. We use 100 samples for the stochastic forward passes and optimise the hyperparameters of the Laplace approximations w.r.t. the cross-entropy on the validation set of MNIST.

We measure the uncertainty of the different methods as the entropy of the

[1] From: http://yaroslavvb.blogspot.nl/2011/09/notmnist-dataset.html

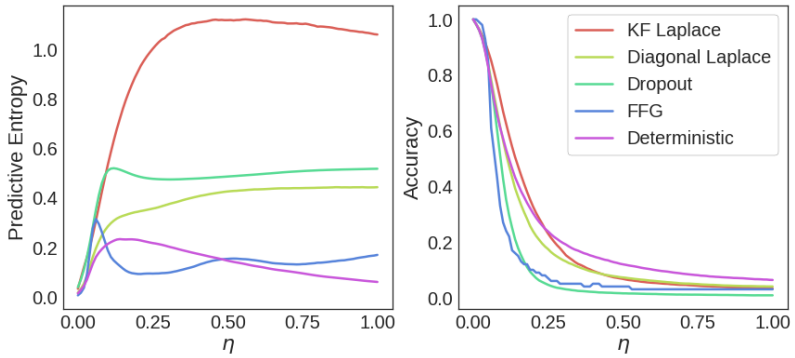


Figure 3.5: Impact of an untargeted adversarial attack with varying step size η on predictive entropy (left) and classification accuracy (right) for different uncertainty estimation method trained on MNIST.

predictive distribution, which has a minimal value of 0 when a single class is predicted with certainty and a maximum of about 2.3 for uniform predictions. Fig. 3.4 shows the inverse empirical cumulative distribution of the entropy values obtained from the four methods. Consistent with the results in [83], averaging the probabilities of multiple passes through the network yields predictions with higher uncertainty than a deterministic pass that approximates the geometric average [317]. However, there still are some images that are predicted to be a digit with certainty. Our Kronecker factored Laplace approximation makes hardly any predictions with absolute certainty and assigns high uncertainty to most of the letters as desired. The diagonal Laplace approximation required stronger regularisation towards predicting deterministically, yet it performs similarly to Dropout. As shown in Tab. 3.2, however, the network makes predictions on the test set of MNIST with similar accuracy to the deterministic forward pass and MC Dropout when using our approximation. The variational factorised Gaussian posterior has low uncertainty as expected.

3.2.5.3 Adversarial examples

To further test the robustness of our prediction method close to the data distribution, we perform an adversarial attack on a neural network. As first demonstrated in [325], neural networks are prone to being fooled by gradient-based changes to their inputs. Li and Gal [205] suggest, and provide empirical support, that Bayesian models may be more robust to such attacks, since they implicitly form an infinitely large

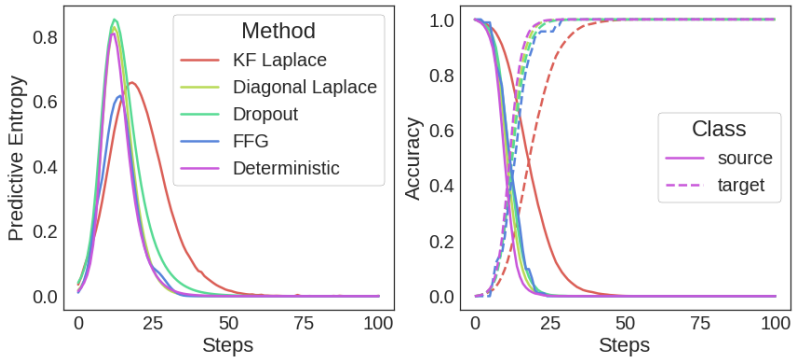


Figure 3.6: Impact of a targeted adversarial attack with varying number of steps on predictive entropy (left) and classification accuracy (right) for different uncertainty estimation method trained on MNIST.

ensemble by integrating over the model parameters. For our experiments, we use the fully connected net trained on MNIST from the previous section and compare the sensitivity of the different prediction methods for two kinds of adversarial attacks.

First, we use the untargeted Fast Gradient Sign method $\mathbf{x}_{adv} = \mathbf{x} - \eta\,\mathrm{sgn}(\nabla_{\mathbf{x}} \max_y \log p^{(M)}(y|\mathbf{x}))$ suggested in [97], which takes the gradient of the class predicted with maximal probability by method M w.r.t. the input $\mathbf{x}$ and reduces this probability with varying step size η. This step size is rescaled by the difference between the maximal and minimal value per dimension in the dataset. It is to be expected that this method generates examples away from the data manifold, as there is no clear subset of the data that corresponds to e.g. "not ones". Also, note that pixel values on grey-scale images increment by a value of $1/255$, so outside of small values of η the change to the data will correspond to perceptible modifications of the data.

Fig. 3.5 shows the average predictive uncertainty and the accuracy on the original class on the MNIST test set as the step size η increases. The Kronecker factored Laplace approximation achieves significantly higher uncertainty than any other prediction method as the images move away from the data. Both the diagonal and the Kronecker factored Laplace maintain higher accuracy than MC Dropout on their original predictions. Interestingly, the deterministic forward pass appears to be most robust in terms of accuracy, however it has much smaller uncertainty on the predictions it makes and will confidently predict a false class for most images,

whereas the other methods are more uncertain.

Furthermore, we perform a targeted attack that attempts to force the network to predict a specific class, in our case '0' following [205]. Hence, for each method, we exclude all data points in the test set that are already predicted as '0'. The updates are of similar form to the untargeted attack, however they increase the probability of the pre-specified class y rather than decreasing the current maximum as $\mathbf{x}_y^{(t+1)} = \mathbf{x}_y^{(t)} + \eta \, \mathrm{sgn}(\nabla_{\mathbf{x}} \log p^{(M)}(y|\mathbf{x}_y^{(t)}))$, where $\mathbf{x}_y^{(0)} = \mathbf{x}$.

We use a step size of $\eta{=}10^{-2}$ for the targeted attack. The uncertainty and accuracy on the original and target class are shown in Fig. 3.6. Here, the Kronecker factored Laplace approximation has slightly smaller uncertainty at its peak in comparison to the other methods, however it appears to be much more robust. It only mis-classifies over 50% of the images after about 20 steps, whereas for the other methods this is the case after roughly 10 steps and reaches 100% accuracy on the target class after almost 50 updates, whereas the other methods are fooled on all images after about 25 steps.

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

In the following, we also show figures for the adversarial experiments in which we calculate the curvature per datapoint and without data augmentation:

In order to study the impact of an accurate estimation of the Hessian corresponding to the training process, Fig. 3.7 and Fig. 3.8 show how the Laplace approximation with the curvature estimated from 1000 randomly sampled binary MNIST images and the activation Hessian calculated with a minibatch size of 100 performs in comparison to the curvature factor being calculated without any data augmentation with a batch size of 100 or exactly. We note that without data augmentation we had to use much stronger regularisation of the curvature factors, in particular we had to add a non-negligible multiple of the identity to the factors, whereas with data

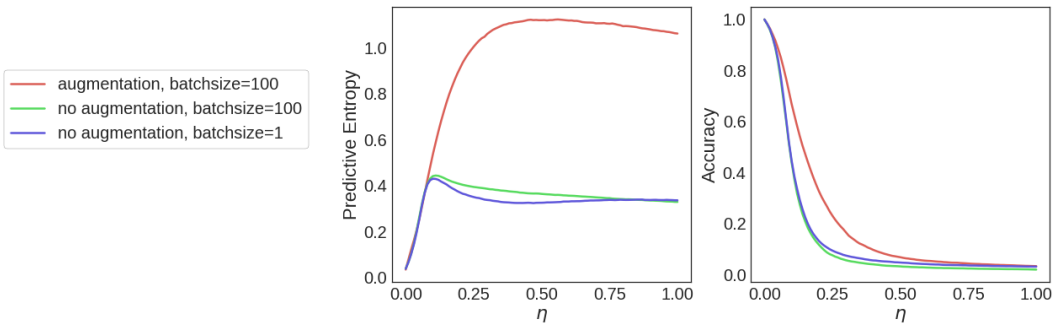


Figure 3.7: Untargeted adversarial attack for Kronecker factored Laplace approximation with the curvature calculated with and without data augmentation/approximating the activation Hessian.

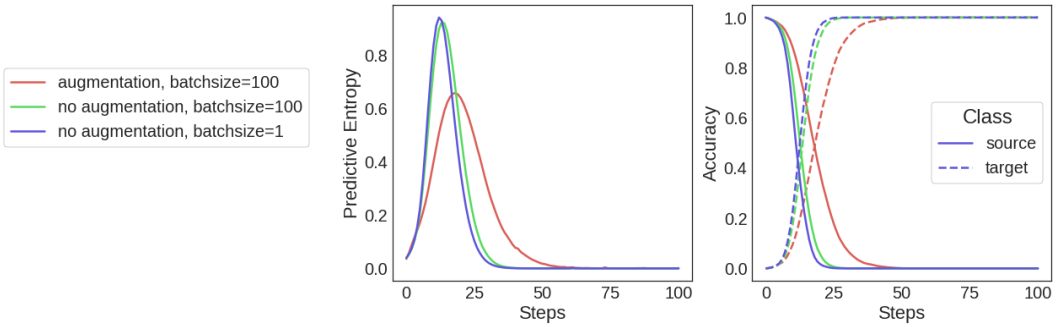


Figure 3.8: Targeted adversarial attack for Kronecker factored Laplace approximation with the curvature calculated with and without data augmentation/approximating the activation Hessian.

augmentation it was only needed to ensure that the matrices are invertible. The Kronecker factored Laplace approximation reaches particularly high uncertainty on the untargeted adversarial attack and is most robust on the targeted attack when using data augmentation, suggesting that it is particularly well suited for large datasets and ones where some form of data augmentation can be applied. The difference between approximating the activation Hessian over a minibatch and calculating it exactly appears to be negligible.

3.2.5.4 Uncertainty on mis-classifications

To highlight the scalability of our method, we apply it to a state-of-the-art convolutional network architecture. Recently, deep residual networks [114, 115] have been the most successful ones among those. As demonstrated in [103], Kronecker factored curvature methods are applicable to convolutional layers by interpreting

them as matrix-matrix multiplications.

We compare our uncertainty estimates on wide residual networks [369], a recent variation that achieved competitive performance on CIFAR100 [178] while, in contrast to most other residual architectures, including Dropout at specific points. While this does not correspond to using Dropout in the Bayesian sense [82], it allows us to at least compare our method to the uncertainty estimates obtained from Dropout.

We note that it is straightforward to incorporate batch normalisation [149] into the curvature backpropagation algorithms, so we apply a standard Laplace approximation to its parameters as well. We are not aware of any interpretation of Dropout as performing Bayesian inference on the parameters of batch normalisation.

Our wide residual network has n=3 block repetitions and a width factor of k=8 on CIFAR100 with and without Dropout using hyperparameters taken from [369]: the network parameters are trained on a cross-entropy loss using Nesterov momentum with an initial learning rate of 0.1 and momentum of 0.9 for 200 epochs with a minibatch size of 128. We decay the learning rate every 50 epochs by a factor of 0.2, which is slightly different to the schedule used in [369] (they decay after 60, 120 and 160 epochs). As the original authors, we use L_2-regularisation with a factor of 5×10^{-4}.

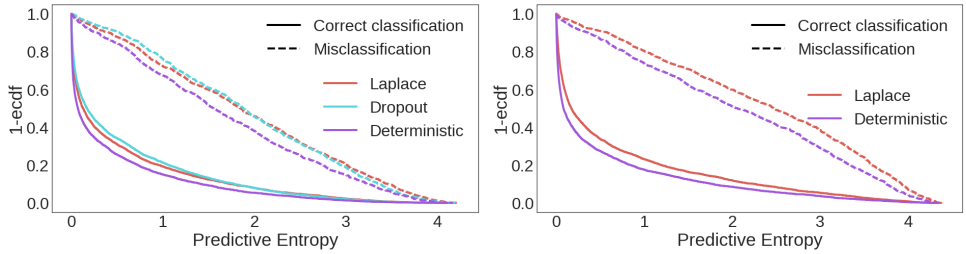
We make one small modification to the architecture: instead of downsampling with 1×1 convolutions with stride 2, we use 2×2 convolutions. This is due to Theano not supporting the transformation of images into the patches extracted by a convolution for 1×1 convolutions with stride greater than 1, which we require for our curvature backpropagation through convolutions.

We apply a standard Laplace approximation to the batch normalisation parameters — a Kronecker factorisation is not needed, since the parameters are one-dimensional. When calculating the curvature factors, we use the moving averages for the per-layer means and standard deviations obtained after training, in order to maintain independence between the data points in a minibatch.

Again, the accuracy of the prediction methods is comparable, see Tab. 3.3.

Table 3.3: Accuracy on the final 5,000 CIFAR100 test images for a wide residual network trained with and without Dropout.

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



(a) Dropout Wide Resnet **(b)** Deterministic Wide Resnet

Figure 3.9: Inverse ecdf of the predictive entropy from Wide Residual Networks trained with and without Dropout on CIFAR100. For mis-classifications, curves on top corresponding to higher uncertainty are desirable, and curves on the bottom for correct classifications.

For calculating the curvature factors, we draw 5,000 samples per image using the same data augmentation as during training, effectively increasing the dataset size to 2.5×10^8. The diagonal approximation had to be regularised to the extent of becoming deterministic, so we omit it from the results.

In Fig. 3.9 we compare the distribution of the predictive uncertainty on the test set.[2] We distinguish between the uncertainty on correct and incorrect classifications, as the mistakes of a system used in practice may be less severe if the network can at least indicate that it is uncertain. Thus, high uncertainty on mis-classifications and low uncertainty on correct ones would be desirable, such that a system could return control to a human expert when it can not make a confident decision. In general, the network tends to be more uncertain on its mis-classifications than its correct ones regardless of whether it was trained with or without Dropout and of the method used for prediction. Both Dropout and the Laplace approximation similarly increase the uncertainty in the predictions, however this is irrespective of the correctness of the classification. Yet, our experiments show that the Kronecker factored Laplace

[2]We use the first 5,000 images as a validation set to tune the hyperparameters of our Laplace approximation and the final 5,000 ones for evaluating the predictive uncertainty on all methods.

approximation can be scaled to modern convolutional networks and maintain good classification accuracy while having similar uncertainty about the predictions as Dropout.

We had to use much stronger regularisation for the Laplace approximation on the wide residual network, possibly because the block-diagonal approximation becomes more inaccurate on deep networks, possibly because the number of parameters is much higher relative to the number of data. It would be interesting to see how the Laplace approximations behaves on a much larger dataset like ImageNet for similarly sized networks, where we have a better ratio of data to parameters and curvature directions. However, even on a relatively small dataset like CIFAR we did not have to regularise the Laplace approximation to the degree of the posterior becoming deterministic. After completion of this work, [9] pointed out that applying the Laplace approximation in networks with normalisation layers, such as ResNets, needs to account for the weight scaling invariance induced by these layers, which may provide an explanation for these results.

3.2.6 Related work

Most recent attempts to approximating the posterior of a neural network use fully-factorised Gaussian distributions within a variational [130, 100, 26, 168, 164, 260, 361, 324] or moment propagation-based framework [124, 90]. These all assume independence between the individual weights which, particularly when optimising the KL divergence, often lets the model underestimate the uncertainty about the weights.

This work is a scalable approximation of [221]. Since the per-layer Hessian of a neural network is infeasible to compute, we suggest a factorisation of the covariance into a Kronecker product, leading to a more efficient *matrix* normal distribution. The Kronecker factor Laplace approximation had concurrently been explored in [99] within the context of a Bayesian interpretation of meta-learning [303, 245, 329, 70] The posterior that we obtain is reminiscent of [214] and [320], who optimise the parameters of a matrix normal distribution as their weights, which requires a modification of the training procedure while the Laplace approximation can be

applied post-hoc to a trained network. Concurrently to this work, another variational approach with a matrix normal per layer has been proposed based on a noisy variant of KFAC [373, 15]. Other approaches that model posterior correlations include [242, 151], which have a low-rank covariance structure across all parameters, and [334], which has a low-rank structure for the posterior of each layer independently.

Following the publication of the present method, various other works have used the Laplace approximation for approximate inference in neural networks Most relevant, [198] proposes a more accurate approximation based on the improved Fisher approximation of [86], which uses a full rather than a Kronecker factored eigenbasis, leading to a multivariate rather than matrix normal distribution per layer. As the eigenbasis of the covariance of the multivariate normals is Kronecker factored, efficient sampling is still possible. The method further includes a diagonal correction term to match the diagonal of the full Fisher, which leads to a need for a low-rank approximation of the Kronecker factors. Other corrections to improve the accuracy of the Kronecker factored approximation have been proposed in [340, 175], but only been evaluated in the context of second order optimisation.

Immer et al. [148] report improved predictive performance based on the linearisation of the predictive posterior by MacKay [221] .which can be implemented efficiently with the flexible automatic differentiation toolbox of JAX [32]. The linearised Laplace approximation has also been investigated in [74] for small networks where no approximations of the curvature are needed. Finally, Immer et al. [147] leverage the Kronecker factored Laplace approximation to tune neural network hyperparameters based on the approximate marginal likelihood.

Kristiadi et al. [176] report that a last-layer Laplace approximation is sufficient to alleviate the overconfidence issues of neural networks with ReLU non-linearities and further demonstrate its efficacy in a closed-form approximation of the predictive distribution for classification in [131]. Similarly, Liu et al. [208] use a Laplace approximation on the last-layer weights as part of a parametric GP approximation with random features [275, 276]. Eschenhagen et al. [63] use the last-layer approximation on all networks in an ensemble and find this to further improve the predictive

uncertainty. Ash et al. [11] also rely on a last-layer approximation of the Fisher, but for uncertainty quantification in active learning. Further, a low-rank Laplace approximation (referred to with its frequentist name as 'Delta method' in the corresponding work) has been proposed by Nilsen et al. [256]. Daxberger et al. [53] approximate the posterior over a subset of the parameters with a Laplace approximation, leaving the others deterministic. Finally, Kristiadi et al. [177] add learnable 'uncertainty units' to a Laplace-approximated BNN and explicitly tune these to predict with high uncertainty on OOD and high confidence on in-distribution data.

Another method that constructs a multivariate Gaussian distribution with low-rank covariance structure over neural network parameters is SWAG [225], however without explicitly approximating the posterior. Rather than setting the mean to the parameters at the end of the optimisation process, it continues to make gradient updates after convergence and averages the parameters. This allows for the mean of the Gaussian to be placed away from the bottom of an asymmetric loss surface, which has been linked to empirically generalising better for deterministic networks [150]. The covariance is then the subspace spanned by parameter values collected at regular intervals. A connection to Bayesian inference could be made through [227], which argues that under certain assumptions and with a learning rate and batch size that makes the sampling noise match the covariance of the mode, SGD can be interpreted as performing variational inference. Experiments in [225] demonstrate, however, that these assumptions are generally not met, so that SWAG might best be interpreted as a pragmatic Laplace approximation. The paper reports that SWAG performs better than the Kronecker factored Laplace approximation for a range of large scale network architectures. However, it is not clear if this performance difference arises from the different location of the mean, the structure of the posterior covariance, the way it is estimated or a combination of (a subset of) these aspects.

3.3 Continual learning

We now discuss discuss how to leverage the previously introduced Laplace approximation to learn from a stream of datasets online. Creating an agent that performs

well across multiple tasks and continuously incorporates new knowledge has been a longstanding goal of research on artificial intelligence. When training on a sequence of tasks, however, the performance of many machine learning algorithms, including neural networks, decreases on older tasks when learning new ones. This phenomenon has been termed 'catastrophic forgetting' [77, 232, 283] and has recently received attention in the context of deep learning [95, 170]. Catastrophic forgetting cannot be overcome by simply initialising the parameters for a new task with optimal ones from the old task and hoping that stochastic gradient descent will stay sufficiently close to the original values to maintain good performance on previous datasets [95].

Bayesian learning provides an elegant solution to this problem. It combines the current data with prior information to find an optimal trade-off in our belief about the parameters. In the sequential setting, such information is readily available: the posterior over the parameters given all previous datasets. It follows from Bayes' rule that we can use the posterior over the parameters after training on one task as our prior for the next one. As the posterior over the weights of a neural network is typically intractable, we need to approximate it. This type of Bayesian online learning has been studied extensively in the literature [259, 87, 138].

In this section, we combine Bayesian online learning [259] with the Kronecker factored Laplace approximation [289] to update a quadratic penalty for every new task. The block-diagonal Kronecker factored approximation of the Hessian [229, 29] allows for an expressive scalable posterior that takes interactions between weights within the same layer into account. In our experiments we show that this principled approximation of the posterior leads to substantial gains in performance over simpler diagonal methods, in particular for long sequences of tasks.

3.3.1 Bayesian online learning

In continual learning, we are interested in finding parameters $\boldsymbol{\theta}$ or a distribution over the for a single neural network to perform well across multiple tasks $\mathcal{D}_1, \ldots, \mathcal{D}_T$. However, the datasets arrive sequentially and we can only train on one of them at a time.

Here, we first discuss how Bayesian online learning solves this problem on a

high-level and then discuss the online Laplace approximation as an instantiation. We continue to incorporate the Kronecker factored approximation of the Hessian of the likelihood into the updating procedure and detail how to efficiently calculate the resulting quadratic penalty and how the approximate posterior can be regularised to further improve performance. Finally, we discuss some ad-hoc variants of the online Laplace approximation that relate to existing methods in the literature.

Bayesian online learning [259], or Assumed Density Filtering [231], is a framework for updating an approximate posterior when data arrive sequentially. Using Bayes' rule we would like to simply incorporate the most recent dataset $\mathcal{D}_{t+1}$ into the posterior as

$$p(\boldsymbol{\theta} \mid \mathcal{D}_{1:t+1}) = \frac{p(\mathcal{D}_{t+1} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta} \mid \mathcal{D}_{1:t})}{\int d\boldsymbol{\theta}' p\left(\mathcal{D}_{t+1} \,\middle|\, \boldsymbol{\theta}'\right) p\left(\boldsymbol{\theta}' \,\middle|\, \mathcal{D}_{1:t}\right)}, \tag{3.48}$$

where we use the posterior $p(\boldsymbol{\theta} \mid \mathcal{D}_{1:t})$ from the previously observed tasks as the prior over the parameters for the most recent task. As the posterior given the previous datasets is typically intractable, Bayesian online learning formulates a parametric approximate posterior q with parameters $\boldsymbol{\phi}_t$, which it iteratively updates in two steps:

Update step In the update step, the approximate posterior q with parameters $\boldsymbol{\phi}_t$ from the previous task is used as a prior to find the new posterior given the most recent data

$$p(\boldsymbol{\theta} \mid \mathcal{D}_{t+1}, \boldsymbol{\phi}_t) = \frac{p(\mathcal{D}_{t+1} \mid \boldsymbol{\theta})\, q(\boldsymbol{\theta} \mid \boldsymbol{\phi}_t)}{\int d\boldsymbol{\theta}' p\left(\mathcal{D}_{t+1} \,\middle|\, \boldsymbol{\theta}'\right) q\left(\boldsymbol{\theta}' \,\middle|\, \boldsymbol{\phi}_t\right)}. \tag{3.49}$$

Projection step The projection step finds the distribution within the parametric family of the approximation that most closely resembles this posterior, i.e. sets $\boldsymbol{\phi}_{t+1}$ such that

$$q\left(\boldsymbol{\theta} \,\middle|\, \boldsymbol{\phi}_{t+1}\right) \approx p(\boldsymbol{\theta} \mid \mathcal{D}_{t+1}, \boldsymbol{\phi}_t). \tag{3.50}$$

Opper and Winther [259] suggest minimising the KL-divergence between the approximate and the true posterior, however this is mostly appropriate for models where the update-step posterior and a solution to the KL-divergence are available in closed form. In the following, we therefore propose using a Laplace approximation to make Bayesian online learning tractable for neural networks.

3.3.2 The online Laplace approximation

Neural networks have found wide-spread success and adoption by performing simple MAP inference, i.e. finding a mode of the posterior

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta} \mid \mathcal{D}) = \arg\max_{\boldsymbol{\theta}} \log p(\mathcal{D} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}), \tag{3.51}$$

where $p(\mathcal{D} \mid \boldsymbol{\theta})$ is the likelihood of the data and $p(\boldsymbol{\theta})$ the prior. Most commonly used loss functions and regularisers fit into this framework, e.g. using a categorical cross-entropy with L_2-regularisation corresponds to modeling the data with a categorical distribution and placing a zero-mean Gaussian prior on the network parameters. A local mode of this objective function can easily be found using standard gradient-based optimisers.

Around a mode, the posterior can be locally approximated using a second-order Taylor expansion, resulting in a Normal distribution with the MAP parameters as the mean and the Hessian of the negative log posterior around them as the precision.

3.3.2.1 Quadratic regulariser & curvature update

We therefore proceed in two iterative steps similar to Bayesian online learning, using a Gaussian approximate posterior for q, such that $\boldsymbol{\phi}_t = \{\boldsymbol{\theta}_t^*, \boldsymbol{\Lambda}_t\}$ consists of a mean $\boldsymbol{\theta}^*$ and a precision matrix $\boldsymbol{\Lambda}$.

Update step As the posterior of a neural network is intractable for all but the simplest architectures, we will work with the unnormalised posterior. The normalisation constant is not needed for finding a mode or calculating the Hessian. The Gaussian approximate posterior results in a quadratic penalty term for the next task, encouraging the new parameters to stay close to the mean of the previous approximate posterior

$$\begin{aligned} -\log p(\boldsymbol{\theta} \mid \mathcal{D}_{t+1}, \boldsymbol{\phi}_t) &\simeq -\log p(\mathcal{D}_{t+1} \mid \boldsymbol{\theta}) - \log q(\boldsymbol{\theta} \mid \boldsymbol{\phi}_t) \\ &\simeq -\log p(\mathcal{D}_{t+1} \mid \boldsymbol{\theta}) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_t^*)^\top \boldsymbol{\Lambda}_t (\boldsymbol{\theta} - \boldsymbol{\theta}_t^*) =: \mathcal{L}_t(\boldsymbol{\theta}). \end{aligned} \tag{3.52}$$

Projection step In the projection step we approximate the posterior with a new multivariate Gaussian.

We first set the mean of the approximation to a maximum of the new posterior

$$\boldsymbol{\theta}^*_{t+1} = \arg\max_{\boldsymbol{\theta}} \log p(\mathcal{D}_{t+1} \mid \boldsymbol{\theta}) + \log q(\boldsymbol{\theta} \mid \boldsymbol{\phi}_t) \tag{3.53}$$

$$= \arg\max_{\boldsymbol{\theta}} \log p(\mathcal{D}_{t+1} \mid \boldsymbol{\theta}) - \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^*_t)^\top \boldsymbol{\Lambda}_t (\boldsymbol{\theta} - \boldsymbol{\theta}^*_t) \tag{3.54}$$

and then perform a quadratic approximation around it, which requires calculating the Hessian of the negative objective. This leads to a recursive update to the precision with the Hessian of the most recent log likelihood, as the Hessian of the negative log approximate posterior is its precision

$$\boldsymbol{\Lambda}_{t+1} = \bar{\boldsymbol{H}}_{t+1}(\boldsymbol{\theta}^*_{t+1}) + \boldsymbol{\Lambda}_t, \tag{3.55}$$

where $\bar{\boldsymbol{H}}_{t+1}(\boldsymbol{\theta}^*_{t+1}) = -\left.\frac{\partial^2 \log p(\mathcal{D}_{t+1} \mid \boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}^*_{t+1}}$ is the Hessian of the negative log likelihood of dataset $t+1$ around the mode after training on dataset $t+1$.

The precision of a Gaussian is required to be positive semi-definite, which is the case for the Hessian at a mode. In order to numerically guarantee this in practice, we use the Fisher Information as an approximation that is positive semi-definite by construction.

The recursion is initialised with the Hessian of the log prior, which is typically constant. For a zero-mean isotropic Gaussian prior, corresponding to an L_2-regulariser, it is simply the identity matrix times the prior precision.[3]

A desirable property of the Laplace approximation is that the approximate posterior becomes peaked around its current mode as we observe more data. This becomes particularly clear if we think of the precision matrix as the product of the number of data points and the average precision. By becoming increasingly peaked, the approximate posterior will naturally allow the parameters to change

[3] Huszár [146] discussed a similar recursive Laplace approximation for online learning, however with limited experimental results and in the context of using a diagonal approximation to the Hessian.

less for later tasks. At the same time, even though the Laplace method is a local approximation, we would expect it to leave sufficient flexibility for the parameters to adapt to new tasks, as the Hessian of neural networks has been observed to be flat in most directions [298].

3.3.2.2 Regularising the approximate posterior

Kirkpatrick et al. [170], who develop a similar method inspired by the Laplace approximation (see Section 3.3.4), termed Elastic Weight Consolidation (EWC), suggest using a multiplier λ on the quadratic penalty in Eq. 3.54. This hyperparameter provides a way of trading off retaining performance on previous tasks against having sufficient flexibility for learning a new one. As modifying the objective would propagate into the recursion for the precision matrix, we instead place the multiplier on the Hessian of each log likelihood and update the precision as

$$\boldsymbol{\Lambda}_{t+1} = \lambda \bar{\boldsymbol{H}}_{t+1}(\boldsymbol{\theta}^*_{t+1}) + \boldsymbol{\Lambda}_t. \tag{3.56}$$

The multiplier affects the width of the approximate posterior and thus the location of the next MAP estimate. As it acts directly on the parameter of a probability distribution, its optimal value can inform us about the quality of our approximation: if it strongly deviates from its natural value of 1, our approximation is a poor one and over- or underestimates the uncertainty about the parameters.

A small λ resulting in high uncertainty shifts the mode towards that of the likelihood, i.e. enables the network to learn the new task well even if our posterior approximation underestimates the uncertainty. Vice versa, increasing λ moves the joint mode towards the prior mode, improving how well the previous parameters are remembered. The optimal choice depends on the true posterior and how closely it is approximated.

In principle, it would be possible to use a different value λ_t for every dataset. In our experiments, we keep the value of λ the same across all tasks as the family of posterior approximation is the same throughout training. Furthermore, using a separate hyperparameter for each task would let the number of hyperparameters

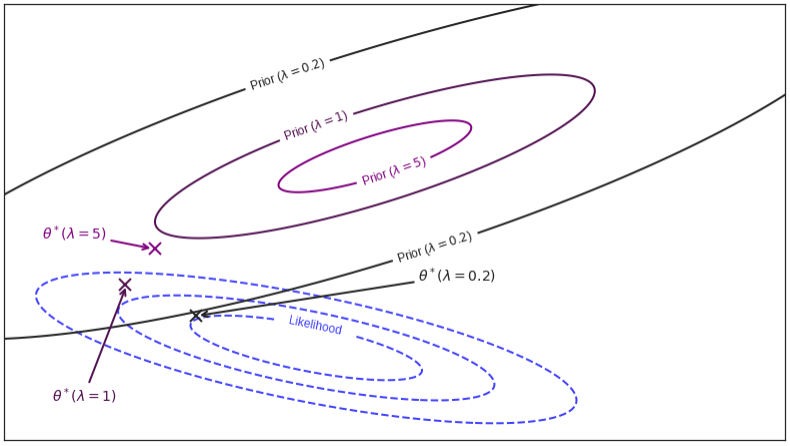


Figure 3.10: Contours of a Gaussian likelihood (dashed blue) and prior (shades of purple) for different values of λ. Values smaller than 1 shift the joint maximum θ^*, marked by a '$\times$',towards that of the likelihood, i.e. the new task, for values greater than 1 it moves towards the prior, i.e. previous tasks.

grow linearly in the number of tasks, which would make tuning them costly.

3.3.3 Kronecker factored approximation

As in the previous section, working with the full Hessian is computationally intractable and we will be using the Kronecker factored approximation that we developed. Hence we approximate the Hessian for each task as block-diagonal

$$\bar{\boldsymbol{H}}_t(\boldsymbol{\theta}_t^*) \approx \text{diag}(\bar{\boldsymbol{H}}_t^{(1)}, \ldots, \bar{\boldsymbol{H}}_t^{(L)}) \tag{3.57}$$

and each block, now dropping the layer index l (which in contrast to other section we had placed as a superscript here) and the dependence on the task parameters to simplify the notation, as Kronecker factored

$$\bar{\boldsymbol{H}}_t = N_t \, \mathbb{E}[\boldsymbol{H}_t] \approx \sqrt{N_t}\, \mathbb{E}[\boldsymbol{\mathcal{Q}}_t] \otimes \sqrt{N_t}\, \mathbb{E}[\boldsymbol{\mathcal{H}}_t] =: \bar{\boldsymbol{\mathcal{Q}}}_t \otimes \bar{\boldsymbol{\mathcal{H}}}_t, \tag{3.58}$$

where N_t denotes the number of data in task t.

3.3.3.1 Updating the Hessian blocks

Plugging the Kronecker factored approximation into the update Eq. 3.55, each block of the precision matrix is

$$\boldsymbol{\Lambda}_{t+1} \approx \bar{\boldsymbol{\mathcal{Q}}}_{t+1} \otimes \bar{\boldsymbol{\mathcal{H}}}_{t+1} + \boldsymbol{\Lambda}_t = \sum_{i=0}^{t+1} \bar{\boldsymbol{\mathcal{Q}}}_i \otimes \bar{\boldsymbol{\mathcal{H}}}_i, \tag{3.59}$$

i.e. a sum of Kronecker products and we define $\boldsymbol{\Lambda}_0$ as the Hessian of the negative prior on the weights, which for a Gaussian prior $\text{vec}(\boldsymbol{W}) \sim \mathcal{N}(\mathbf{0}, \tau^{-1}\boldsymbol{I}) \Leftrightarrow \boldsymbol{W} \sim \mathcal{MN}(\mathbf{0}, \tau^{-\frac{1}{2}}\boldsymbol{I}, \tau^{-\frac{1}{2}}\boldsymbol{I})$ is Kronecker factored.

As Kronecker products do not add up pairwise, one option, which we implement in this work, is to calculate one penalty term per previously seen task Section 3.3.3.2, as commonly done in the literature (Section 3.3.4). If a single term is preferable, we can further approximate each block of the precision at time step t as

$$\boldsymbol{\Lambda}_t \approx \tilde{\boldsymbol{\mathcal{Q}}}_t \otimes \tilde{\boldsymbol{\mathcal{H}}}_t, \tag{3.60}$$

with

$$\tilde{\boldsymbol{\mathcal{Q}}}_t := \frac{\sqrt{N}}{t} \sum_{i=0}^{t} \mathbb{E}\left[\boldsymbol{\mathcal{Q}}_i\right] \quad \text{and} \quad \tilde{\boldsymbol{\mathcal{H}}}_t := \frac{\sqrt{N}}{t} \sum_{i=0}^{t} \mathbb{E}\left[\boldsymbol{\mathcal{H}}_i\right]. \tag{3.61}$$

Here, $N = \sum_{i=1}^{t} N_t$ is the total number of data points observed across all tasks. Hence we would maintain an average of each factor across time and then scale it to the correct magnitude in calculating the penalty.

3.3.3.2 Efficiently computation of the quadratic penalty

Due to the block-diagonal approximation, the quadratic penalty decomposes across layers as

$$\mathcal{L}_t(\theta) = \boldsymbol{\delta}^\top \boldsymbol{\Lambda}_t \boldsymbol{\delta} \approx \sum_{l=1}^{L} \boldsymbol{\delta}_l^\top \boldsymbol{\Lambda}_t^{(l)} \boldsymbol{\delta}_l, \tag{3.62}$$

where we define $\boldsymbol{\delta} = \boldsymbol{\theta} - \boldsymbol{\theta}_t^*$ and $\boldsymbol{\delta}_l$ the corresponding part of layer l.

Again dropping all layer indices, with the Kronecker factored approximation

the penalty for each layer becomes

$$\mathcal{L}_t(\boldsymbol{\theta}) \approx \boldsymbol{\delta}^\top \left(\sum_{i=0}^{t} \bar{\boldsymbol{\mathcal{Q}}}_i \otimes \bar{\boldsymbol{\mathcal{H}}}_i \right) \boldsymbol{\delta} = \sum_{i=0}^{t} \boldsymbol{\delta}^\top \left(\bar{\boldsymbol{\mathcal{Q}}}_i \otimes \bar{\boldsymbol{\mathcal{H}}}_i \right) \boldsymbol{\delta}, \tag{3.63}$$

or similarly with the single Kronecker product approximation

$$\mathcal{L}_t(\boldsymbol{\theta}) \approx \boldsymbol{\delta}^\top \left(\tilde{\boldsymbol{\mathcal{Q}}}_t \otimes \tilde{\boldsymbol{\mathcal{H}}}_t \right) \boldsymbol{\delta}. \tag{3.64}$$

In either case, we need to compute a quadratic form over a Kronecker product, which can be done efficiently as

$$\boldsymbol{\delta}^\top (\boldsymbol{\mathcal{Q}} \otimes \boldsymbol{\mathcal{H}}) \boldsymbol{\delta} = \boldsymbol{\delta}^\top \operatorname{vec}(\boldsymbol{\mathcal{H}} \boldsymbol{\Delta} \boldsymbol{\mathcal{Q}}), \tag{3.65}$$

where we define $\operatorname{vec}(\Delta) := \operatorname{vec}(\boldsymbol{W} - \boldsymbol{W}_t^*) = \delta$. We can express this more compactly with the Cholesky decompositions of $\boldsymbol{\mathcal{Q}}$ and $\boldsymbol{\mathcal{H}}$ as

$$\begin{aligned}
\boldsymbol{\delta}^\top (\boldsymbol{\mathcal{Q}} \otimes \boldsymbol{\mathcal{H}}) \boldsymbol{\delta} &= \boldsymbol{\delta}^\top \left(\boldsymbol{L}_{\boldsymbol{\mathcal{Q}}} \boldsymbol{L}_{\boldsymbol{\mathcal{Q}}}^\top \otimes \boldsymbol{L}_{\boldsymbol{\mathcal{H}}} \boldsymbol{L}_{\boldsymbol{\mathcal{H}}}^\top \right) \boldsymbol{\delta} && (3.66) \\
&= \boldsymbol{\delta}^\top \left(\boldsymbol{L}_{\boldsymbol{\mathcal{Q}}} \otimes \boldsymbol{L}_{\boldsymbol{\mathcal{H}}} \right) \left(\boldsymbol{L}_{\boldsymbol{\mathcal{Q}}}^\top \otimes \boldsymbol{L}_{\boldsymbol{\mathcal{H}}}^\top \right) \boldsymbol{\delta} && (3.67) \\
&= \boldsymbol{\delta}^\top \left(\boldsymbol{L}_{\boldsymbol{\mathcal{Q}}} \otimes \boldsymbol{L}_{\boldsymbol{\mathcal{H}}} \right) \left(\boldsymbol{L}_{\boldsymbol{\mathcal{Q}}} \otimes \boldsymbol{L}_{\boldsymbol{\mathcal{H}}} \right)^\top \boldsymbol{\delta} && (3.68) \\
&= \operatorname{vec}\left(\boldsymbol{L}_{\boldsymbol{\mathcal{H}}} \boldsymbol{\Delta} \boldsymbol{L}_{\boldsymbol{\mathcal{Q}}}^\top \right)^\top \operatorname{vec}\left(\boldsymbol{L}_{\boldsymbol{\mathcal{H}}} \boldsymbol{\Delta} \boldsymbol{L}_{\boldsymbol{\mathcal{Q}}}^\top \right) && (3.69) \\
&=: \tilde{\boldsymbol{\delta}}^\top \tilde{\boldsymbol{\delta}} = ||\tilde{\boldsymbol{\delta}}||_2^2 && (3.70)
\end{aligned}$$

3.3.3.3 Computational complexity

Our method requires calculating the expectations of the two Kronecker factors from Eq. 3.11 over the data of the most recent task after training on it as well as calculating the quadratic penalty in Eq. 3.54 using the identity in Eq. 3.65 for every parameter update.

Calculating the Kronecker factors can efficiently be mini-batched and requires the same calculations as a forward and backward pass through the network plus two additional matrix-matrix products. The overall cost is thus effectively equivalent to

that of an extra training epoch. See [229] for more details.

The computational complexity of calculating the quadratic penalty is dominated by the two matrix-matrix products in Eq. 3.65. Assuming that all L layers of the network as well as the inputs and outputs are of dimensionality d, all weight matrices as well as the Kronecker factors will be of dimensionality $d \times d$. In general, the size of the first factor is square in the dimensionality of the input to a layer and that of the second factor square in the number of units, i.e. $d_{in} \times d_{in}$ and $d_{out} \times d_{out}$ for a $d_{out} \times d_{in}$ weight matrix. The complexity of calculating the penalty for all layers is then $\mathcal{O}(Ld^3)$.

Finally, we note that sums of Kronecker products do not add up pairwise, i.e. $\boldsymbol{A} \otimes \boldsymbol{B} + \boldsymbol{C} \otimes \boldsymbol{D} \neq (\boldsymbol{A} + \boldsymbol{C}) \otimes (\boldsymbol{B} + \boldsymbol{D})$, so the corresponding Kronecker factors of different tasks do not simply add. In our implementation, we keep an approximate Hessian for every task in memory, similar to how EWC [170] keeps the MAP parameters for each task. If constant scaling in the number of tasks is required, one can make a further approximation by adding up the Kronecker factors separately. This would be comparable to the independence assumption between the factors within the same task.

3.3.4 Ad-hoc approximations of the penalty

Besides maintaining a principled approximation to the posterior, various modifications of a quadratic-penalty based framework as presented above are imaginable. We consider these as baselines for our experiments. Below we discuss one option based on EWC [170], which maintains a penalty term towards the parameters for each task, and one semi-online option which updates the Hessian to that that around the most recent parameters for all datasets.

3.3.4.1 Per-task Laplace

Following the EWC objective, we could set up an independent regulariser for each previous task, where we carry forward the parameters after training on each dataset and add a quadratic penalty term towards those parameter values under the corresponding Hessian. With no specific assumption as to the approximation of the

Hessian (EWC uses a diagonal one), the penalty term can be denoted in our notation as

$$\sum_{i=1}^{t}(\boldsymbol{\theta}-\boldsymbol{\theta}_i^*)^\top \bar{\boldsymbol{H}}_i(\boldsymbol{\theta}_i^*)(\boldsymbol{\theta}-\boldsymbol{\theta}_i^*). \tag{3.71}$$

This is reminiscent of approximating a posterior independently per-task using only the likelihood Hessian. Hence we label this approach 'per-task Laplace' in the following. As discussed in [146], this may over-regularise towards tasks observed early on, as the parameter values of a task are implicitly contained in the parameters for the subsequent tasks due to the quadratic regulariser.

3.3.4.2 Approximate Laplace

Further, we will compare to fitting the true posterior with a new Gaussian at every task for which we compute the Hessian of all tasks around the most recent MAP estimate:

$$\boldsymbol{\Lambda}_{t+1} = \boldsymbol{H}_{prior} + \sum_{i=1}^{t+1}\boldsymbol{H}_i(\boldsymbol{\theta}_{t+1}^*) \tag{3.72}$$

This procedure differs from the online Laplace approximation only in evaluating all Hessians at the most recent MAP parameters instead of the respective task's ones. Technically, this is not a valid Laplace approximation, as we only optimise an approximation to the posterior. Hence the optimal parameters for the approximate objective will not exactly correspond to a mode of the true posterior. However, as we will use a positive semi-definite approximation to the Hessian, this will only introduce a small additional approximation error.

Calculating the Hessian across all datasets requires relaxing the sequential learning setting to allowing access to previous data 'offline', i.e. between tasks. We use this baseline to check if there is any loss of information in using estimates of the curvature at previous parameter values.

3.3.5 Experiments

In our experiments we compare our online Laplace approximation to the approximate Laplace approximation of Eq. 3.72 as well as EWC [170] and Synaptic Intelligence (SI) [371], both of which also add quadratic regularisers to the objective. Further, we

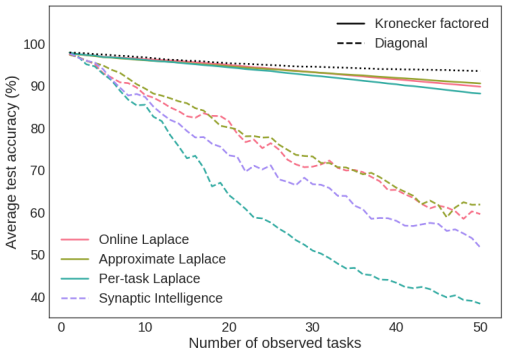
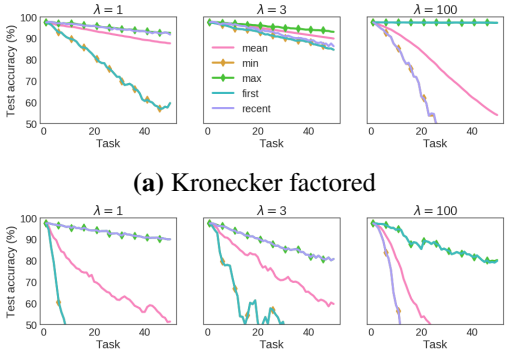


Figure 3.11: Mean test accuracy on a sequence of permuted MNIST datasets. We categorise SI as a diagonal method, as it does not account for parameter interactions. The dotted black line shows the performance of a single network trained on all observed data at each task.



(a) Kronecker factored

(b) Diagonal

Figure 3.12: Effect of λ for different curvature approximations for permuted MNIST. Each plot shows the mean, minimum and maximum across the tasks observed so far, as well as the accuracy on the first and most recent task.

investigate the effect of using a block-diagonal Kronecker factored approximation to the curvature over a diagonal one. We also run EWC with a Kronecker factored approximation, even though the original method is based on a diagonal one.

3.3.5.1 Permuted MNIST

As a first experiment, we test on a sequence of permutations of the MNIST dataset [194]. Each instantiation consists of the 28×28 grey-scale images and labels from the original dataset with a fixed random permutation of the pixels. This makes the individual data distributions mostly independent of each other, testing the ability of each method to fully utilise the model's capacity.

We train a feed-forward network with two hidden layers of 100 units and ReLU non-linearities on a sequence of 50 versions of permuted MNIST. Every one of these datasets is equally difficult for a fully connected network due to its permutation invariance to the input. We stress that our network is smaller than in previous works as the limited capacity of the network makes the task more challenging. Further, we train on a longer sequence of datasets.

Fig. 3.11 shows the mean test accuracy as new datasets are observed for the optimal hyperparameters of each method. We refer to the online Laplace approximation as 'Online Laplace', to the Laplace approximation around an approximate mode

as 'Approximate Laplace' and to adding a quadratic penalty for every set of MAP parameters as in [170] as 'Per-task Laplace'. The per-task Laplace method with a diagonal approximation to the Hessian corresponds to EWC.

We find our online Laplace approximation to maintain higher test accuracy throughout training than placing a quadratic penalty around the MAP parameters of every task, in particular when using a simple diagonal approximation to the Hessian. However, the main difference between the methods lies in using a Kronecker factored approximation of the curvature over a diagonal one.[4] Using this approximation, we achieve over 90% average test accuracy across 50 tasks, almost matching the performance of a network trained jointly on all observed data. Recalculating the curvature for each task instead of retaining previous estimates does not significantly affect performance.

Beyond simple average performance, we investigate different values of the hyperparameter λ on the permuted MNIST sequence of datasets for our online Laplace approximation. The goal is to visualise how it affects the trade-off between remembering previous tasks and being able to learn new ones for the two approximations of the curvature that we consider. Fig. 3.12 shows various statistics of the accuracy on the test set for the smallest and largest value of the hyperparameter on the quadratic penalty that we tested, as well as the one that optimises the validation error.

We are particularly interested in the performance on the first dataset and the most recent one, as a measure for memory and flexibility respectively. For all displayed values of the hyperparameter, the Kronecker factored approximation (Fig. 3.12a) has higher test accuracy than the diagonal approximation (Fig. 3.12b) on both the most recent and the first task, as well as on average. For the natural choice of $\lambda = 1$ (leftmost subfigure respectively), the network's performance decays for the first task for both curvature approximations, yet it is able to learn the most recent task well. The performance on the first task decays more slowly, however, for the more expressive Kronecker factored approximation of the curvature. Increasing the hyperparameter, corresponding to making the prior more narrow as discussed in

[4] In earlier work, e.g. [170, 371], diagonal approximations were reported to be effective for a smaller number of tasks and with substantially larger networks than in our experiments.

Section 3.3.2.2, leads to the network remembering the first task much better at the cost of not being able to achieve optimal performance on the most recently added task. Using $\lambda = 3$ (central subfigure), the value that achieves optimal validation error in our experiments, the Kronecker factored approximation leads to the network performing similarly on the most recent and first tasks. This coincides with optimal average test accuracy. We are not able to find such an ideal trade-off for the diagonal Hessian approximation, resulting in worse average performance and suggesting that the posterior cannot be matched well without accounting for interactions between the weights. Using a large value of $\lambda = 100$ (rightmost subfigure) reverts the order of performance between the most recent and the first task for both approximations: while for small λ the first task is 'forgotten', the network's performance now stays at a high level — for the Kronecker factored approximation it remembers it perfectly — which comes at the cost of being unable to learn new tasks well.

We conclude from our results that the online Laplace approximation overestimates the uncertainty in the approximate posterior about the parameters for the permuted MNIST task, in particular with a diagonal approximation to the Hessian. Overestimating the uncertainty leads to a need for regularisation in the form of reducing the width of the approximate posterior, as the value that optimises the validation error is $\lambda = 3$. Only when regularising too strongly the approximate posterior underestimates the uncertainty about the weights, leading to reduced performance on new tasks for large values of λ. Using a better approximation to the posterior leads to a drastic increase in performance and a reduced need for regularisation in the subsequent experiments. We note that some regularisation is still necessary, suggesting that even the Kronecker factored approximation overestimates the variance in the posterior, and a better approximation could lead to further improvements. However, it is also possible that the Laplace approximation as such requires a large amount of data to estimate the interaction between the parameters sufficiently well; hence it might be best suited for settings where plenty of data are available.

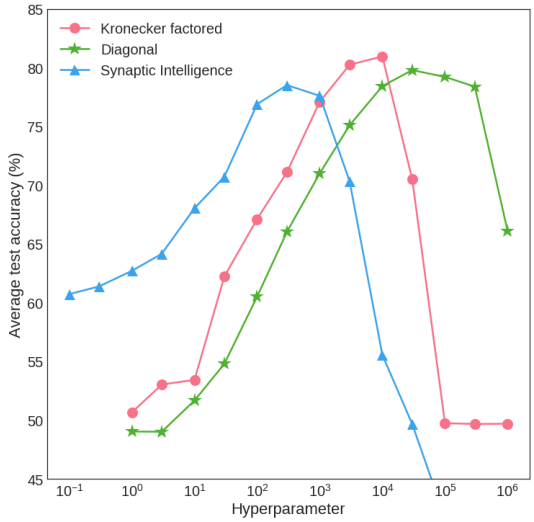


Figure 3.13: Disjoint MNIST test accuracy for the Laplace approximation (hyperparameter: λ) and SI (hyperparameter: c). 'Kronecker factored' and 'Diagonal' refer to the respective curvature approximation for the Laplace method.

3.3.5.2 Disjoint MNIST

We further experiment with the disjoint MNIST task, which splits the MNIST dataset into one part containing the digits '0' to '4', and a second part containing '5' to '9' and training a ten-way classifier on each set separately. Previous work [199] has found this problem to be challenging for EWC, as during the first half of training the network is encouraged to set the bias terms for the second set of labels to highly negative values. This setup makes it difficult to balance out the biases for the two sets of classes after the first task without overcorrecting and setting the biases for the first set of classes to highly negative values. Lee et al. [199] report just over 50% test accuracy for EWC, which corresponds to either completely forgetting the first task or being unable to learn the second one, as each task individually can be solved with around 99% accuracy.

We use an identical network architecture to the previous section and found stronger regularisation of the approximate posterior to be necessary. For the Laplace methods, we tested values of $\lambda \in \{1, 3, 10, \ldots, 3\times10^5, 10^6\}$, and $c \in \{0.1, 0.3, 1, \ldots, 3\times10^4, 10^5\}$ for SI. We train using Nesterov momentum with a learning rate of 0.1 and momentum of 0.9 and decay the learning rate by a factor of 10 every 1000 parameter updates using a batch size of 250. We decay the initial

learning rate for the second task depending on the hyperparameter to prevent the objective from diverging. We test various decay factors for each hyperparameter, but as a rule of thumb found $\frac{\lambda}{10}$ to perform well for the Kronecker factored, and $\frac{\lambda}{1000}$ for the diagonal approximation. The results are averaged across ten independent runs.

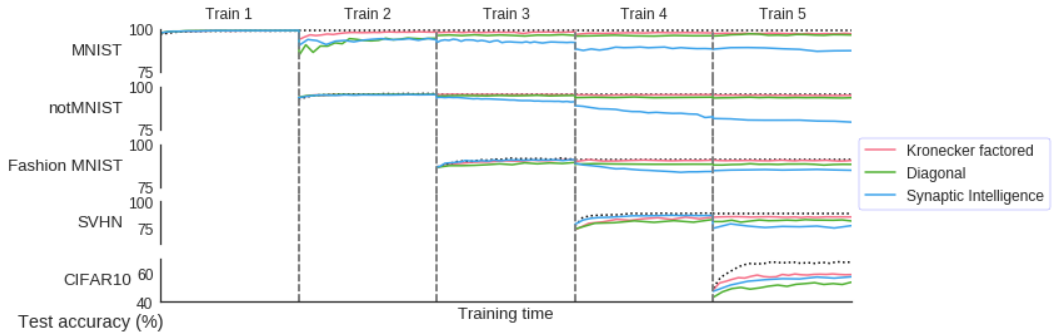
Fig. 3.13 shows the test accuracy for various hyperparameter values for a Kronecker factored and a diagonal approximation of the curvature as well as SI. As there are only two datasets, the three Laplace-based methods are identical, therefore we focus on the impact of the curvature approximation. Approximating the Hessian with a diagonal corresponds to EWC. While we do not match the performance of the method developed in [199], we find the Laplace approximation to work significantly better than reported by the authors. The Kronecker factored approximation gives a small improvement over the diagonal one and requires weaker regularisation, which further suggests that it better fits the true posterior. It also outperforms SI.

3.3.5.3 Vision datasets

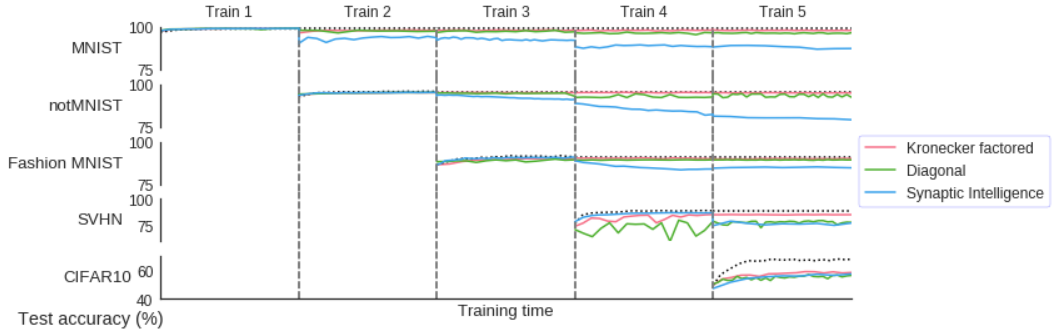
As a final experiment, we test our method on a suite of related vision datasets. Specifically, we train and test on MNIST [194], notMNIST[5], Fashion MNIST [363], SVHN [253] and CIFAR10 [178] in this order. All five datasets contain around $50,000$ training images from 10 different classes. MNIST contains hand-written digits from '0' to '9', notMNIST the letters 'A' to 'J' in different computer fonts, Fashion MNIST different categories of clothing, SVHN the digits '0' to '9' on street signs and CIFAR10 ten different categories of natural images. We zero-pad the images of the MNIST-like datasets to be of size 32×32 and replicate their intensity values over three channels, such that all images have the same format.

We train a LeNet-like architecture [194] with two convolutional layers with 5×5 convolutions with 20 and 50 channels respectively and a fully connected hidden layer with 500 units. We use tanh non-linearities and perform a 2×2 max-pooling operation after each convolutional layer with stride 2. An extension of the Kronecker factored curvature approximations to convolutional neural networks is presented

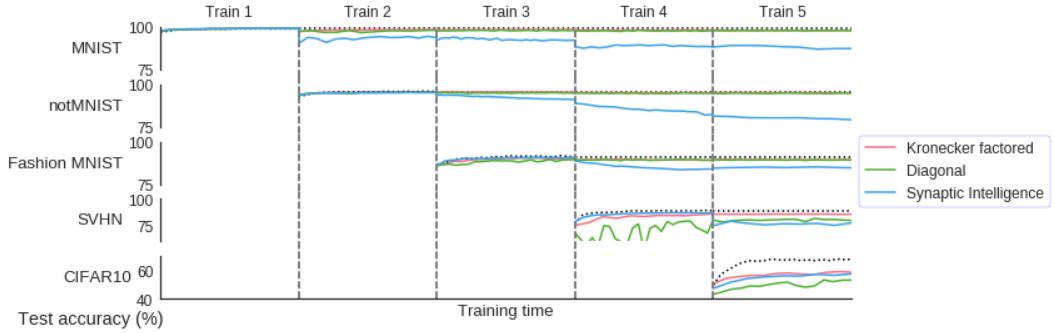
[5]Originally published at www.yaroslavvb.blogspot.co.uk/2011/09/notmnist-dataset.html and downloaded from www.github.com/davidflanagan/notMNIST-to-MNIST



(a) Online Laplace



(b) Approximate Laplace



(c) Per-task Laplace

Figure 3.14: Test accuracy of a convolutional network on a sequence of vision datasets. We train on the datasets separately in the order displayed from top to bottom and show the network's accuracy on each dataset once training on it has started. The dotted black line indicates the performance of a network with the same architecture trained separately on the task. The diagonal and Kronecker factored approximation to the Hessian use the Laplace updates as indicated below the corresponding subfigure.

in [103]. As the meaning of the classes in each dataset is different, we keep the weights of the final layer separate for each task. We optimise the networks as in the permuted MNIST experiment and compare to five baseline networks with the same architecture trained on each task separately.

Overall, the online Laplace approximation in conjunction with a Kronecker

factored approximation of the curvature achieves the highest test accuracy across all five tasks, see Fig. 3.14 and Appendix D.1 for the numerical results. The difference between the three Laplace-based methods is small in comparison to the improvement stemming from the better approximation to the Hessian.

Using a diagonal Hessian approximation for the Laplace approximation, the network mostly remembers the first three tasks, but has difficulties learning the fifth one. SI, in contrast, shows decaying performance on the initial tasks, but learns the fifth task almost as well as our method with a Kronecker factored approximation of the Hessian. However, using the Kronecker factored approximation, the network achieves good performance relative to the individual networks across all five tasks. In particular, it remembers the easier early tasks almost perfectly while being sufficiently flexible to learn the more difficult later tasks better than the diagonal methods, which suffer from forgetting.

3.3.5.4 Optimisation details

For the permuted MNIST experiment, we found the performance of the methods that we compared to mildly depend on the choice of optimiser. Therefore, we optimise all techniques with Adam [166] for 20 epochs per dataset and a learning rate of 10^{-3} as in [371], SGD with momentum [273] with an initial learning rate of 10^{-2} and 0.95 momentum, and Nesterov momentum [252] with an initial learning rate of 0.1, which we divide by 10 every 5 epochs, and 0.9 momentum. For the momentum based methods, we train for at least 10 epochs and early-stop once the validation error does not improve for 5 epochs. Furthermore, we decay the initial learning rate with a factor of $\frac{1}{1+kt}$ for the momentum-based optimisers, where t is the index of the task and k a decay constant. We set k using a coarse grid search for each value of the hyperparameter λ in order to prevent the objective from diverging towards the end of training, in particular with the Kronecker factored curvature approximation. For the Laplace approximation based methods, we consider $\lambda \in \{1, 3, 10, 30, 100\}$; for SI we try $c \in \{0.01, 0.03, 0.1, 0.3, 1\}$. We ultimately pick the combination of optimiser, hyperparameter and decay rate that gives the best validation error across all tasks at the end of training. For the Laplace-based methods, we found momentum based

optimisers to lead to better performance, whereas Adam gave better results for SI.

3.3.6 Related work

Our method builds on Bayesian online learning [259] and Laplace propagation [64]. In contrast to Bayesian online learning, as we cannot update the posterior over the weights in closed form, we use gradient-based methods to find a mode and perform a quadratic approximation around it, resulting in a Gaussian approximation. Laplace propagation, similar to expectation propagation [238], maintains a factor for every task, but approximates each of them with a Gaussian. It performs multiple updates, whereas we use each dataset only once to update the approximation to the posterior.

The most similar method to ours for overcoming catastrophic forgetting is EWC [170], in particular its concurrent online variant [305]. EWC approximates the posterior after the first task with a diagonal Gaussian. However, it continues to add a penalty for every new task [171]. This is more closely related to Laplace propagation, but may be overcounting early tasks [146] and does not approximate the posterior. Furthermore, EWC uses a simple diagonal approximation to the Hessian. Lee et al. [199] approximate the posterior around the mode for each dataset with a diagonal Gaussian in addition to a similar approximation of the overall posterior. They update this approximation to the posterior as the Gaussian that minimises the KL divergence with the individual posterior approximations. Nguyen et al. [254] and its natural gradient variant [341] implement online variational learning [87, 138], which fits an approximation to the posterior through the variational lower bound and then uses this approximation as the prior on the next task. Their Gaussian is fully factorised, hence they do not take weight interactions into account either. [2] extends the framework to share parts of the network between tasks in a learnable manner. [212] describes how tempering the KL term in the ELBO connects variational continual learning and the online Laplace approximation. This section links the Kronecker factored Laplace approximation [289] to Bayesian online learning [259] similar to how Variational Continual Learning [254] connects Online Variational Learning [87, 138] to Bayes-by-Backprop [26].

After completion of this work, [39] explores the online Laplace framework

with a low rank approximation of the Hessian, but does not achieve performance improvements over the Kronecker factored approximation. [197] uses the more accurate Kronecker factored Fisher approximation of [86] and incorporates batch normalisation. [204] proposes to make the Hessian more amenable to approximation through the use of 'linear sketching methods'. In a similar spirit to the present work, [210] proposes rotated EWC to overcome the overly strong diagonal approximation of the curvature and similarly finds performance improvements. [367] confirms our results of the Kronecker factored approximation improving over the diagonal one on some additional benchmark problems. [158] further combines the Kronecker factored online Laplace approximation with projected gradient methods for continual learning in recurrent neural networks. [209] investigates the online Laplace approximation in a federated learning setting. [233] applies the method proposed in this section to continual learning with large-scale language models.

[241] suggests that flat minima – which can be targeted by appropriately setting mini batch size and learning rate [162, 134, 154, 376] and have previously been argued to improve generalisation [223, 132] – can reduce forgetting and that regularisation-based online learning approaches typically set hyperparameters accordingly. It would be interesting to investigate whether this is strictly due to such minima generalising better and being more robust to forgetting, or whether there are additional factors at play, e.g. the quadratic approximation of the Laplace approximation being more accurate.

Rather than placing priors on the weights, [333, 264] propose a functional regulariser that encourages the predictions on a subset of previous data to remain similar, rather than maintaining the parameters values. The former achieve this by replacing the output layer of the network with a Gaussian process [281] and using sparse approximations [314, 331, 122], while the latter exploit the equivalence between BNNs with Gaussian approximate posteriors and Gaussian processes [165] to select a subset of particularly relevant training data.

[142, 66] discuss evaluation of continual learning methods and [173, 201] cover limitations of regularisation-based continual learning.

Various methods for overcoming catastrophic forgetting without a Bayesian motivation have also been proposed over the past year. Zenke et al. [371] develop 'Synaptic Intelligence' (SI), another quadratic penalty on deviations from previous parameter values where the importance of each weight is heuristically measured as the path length of the updates on the previous task. Lopez-Paz and Ranzato [213] formulate a quadratic program to project the gradients such that the gradients on previous tasks do not point in a direction that decreases performance; however, this requires keeping some previous data in memory. Shin et al. [310] suggest a dual architecture including a generative model that acts as a memory for data observed in previous tasks. Other approaches that tackle the problem at the level of the model architecture include [296], which augments the model for every new task, and [69], which trains randomly selected paths through a network. Serrà et al. [307] propose sharing a set of weights and modifying them in a learnable manner for each task. He and Jaeger [116] introduce conceptor-aided backpropagation to shield gradients against reducing performance on previous tasks.

3.4 Conclusion

We presented a scalable approximation to the Laplace approximation for the posterior of a neural network and provided experimental results suggesting that the uncertainty estimates are on par with popular alternatives like MC Dropout, if not better. The Laplace approximation enables practitioners to obtain principled uncertainty estimates from their models, even if they were trained in a maximum likelihood/MAP setting. We further leveraged the approximation for Bayesian online learning method to reduce forgetting in neural networks. By formulating a principled approximation to the posterior and taking interactions between the parameters into account, we were able to substantially improve over EWC [170] and SI [371], two popular methods that also add a quadratic regulariser to the objective for new tasks.

Our continual learning results have been independently confirmed in [367] and various extensions of our Laplace approximation have been proposed since its publication, such as using more accurate Fisher approximations [198] as well as

tuning the prior precision via the marginal likelihood during training rather than via cross-validation afterwards [147] and predicting under a linearisation of the model [148] as originally proposed in [221]. The methods developed in this chapter have been implemented and made available independently as part of the recent 'Laplace Redux' library [52] for PyTorch. Overall, the papers based on the work presented in this section have been followed by a revival of interest in Laplace-based inference in neural networks.

Future directions for this work include developing approximations of the curvature for other layer types than fully connected and convolutional ones, such as recurrent layers as in LSTMs [133] or attention layers in transformer architectures [344]. Further, overcoming the factorisation assumption across layers, e.g. by augmenting the covariance with a low-rank component across all parameters, or developing more accurate approximations of the diagonal blocks of the curvature matrix may further improve performance by representing the posterior more faithfully.

Chapter 4

Parameter-efficient posterior approximations

Rather than increasing the expressivity of the approximate posterior to account for parameter correlations, this chapter investigates methods for reducing the computational burden of BNNs, specifically the storage cost of the parameters. Generally, BNNs increase the number of parameters compared to deterministic networks, as even a variational factorised Gaussian parameters doubles the parameter count with its mean and variance per network parameter.

Section 4.1 proposes a variational inference and an ad-hoc method for *binary* neural networks where each weight is constrained to be -1 or 1. This implies working with Bernoulli distributions which – while still requiring a real-valued probability parameter per weight – allow for storing a number of samples from the posterior at reduced cost, as each binary weight requires only a single bit for storage, whereas real valued weights are typically stored at single precision i.e. 32 bits. On an orthogonal note, Gaussian mean-field networks have been observed to suffer from underfitting issues, making binary neural networks an interesting alternative model class for studying the behaviour of approximate inference methods.

Taking inspiration from sparse approaches for Gaussian processes, Section 4.2 proposes *inducing weights*, an efficient lower-dimensional parameterisation of variational Gaussian posteriors over neural network weights. The method is based on augmenting a matrix Gaussian prior with a smaller inducing weight matrix such that

the marginal prior on the weights remains the same, parameterise the conditional distribution of the weights given the inducing weights and share it between the prior and approximate posterior. To enable efficient sampling from the conditional distribution, we develop an extension of Matheron's rule for matrices, exploiting the Kronecker structure of the augmented joint distribution.

4.1 Bayesian binary neural networks

Neural networks have been applied with increasing success to perceptual data, such as images [54] and speech [342] as models have been growing in size. The simultaneous proliferation of mobile devices has led to a need for approaches to deploy such models under hardware constraints. One particularly promising direction is representing model parameters with reduced precision, down to a single bit per weight [48]. This leads to a 32-fold memory saving compared to the standard full-precision representation of floating point numbers. Binary weights further remove the need for multiplication, reducing computation time as well. Indeed, Rastegari et al. [282] report a 58-fold speedup for their implementation.

Most research efforts so far have focused on training a single network with fixed binary weights [48, 49, 5]. However, simple deterministic networks are known to be poorly calibrated [105] and suffer from catastrophic forgetting [232, 77, 95]. Some recent works [309, 271] have investigated training binary neural networks using distributions over the weights. Their main focus though has still been obtaining a single network with strong predictive performance with a limited evaluation of the quality of the uncertainty estimates, and they do not adopt a Bayesian viewpoint on their training process.

In the continuous weight setting, BNNs have so far mostly been dismissed by the greater deep learning community due to not matching the classification accuracy of deterministic networks in large scale settings, although there has been some progress recently [260]. Most components of BNNs have been suspected to be responsible for this unsatisfying performance, ranging from the prior [354] over factorised Gaussian posteriors [73] and the variational lower bound [339] to the noise

introduced by sampling the weights during optimisation [361, 67]. Since binary neural networks operate on discrete variables or the unit interval, there is more flexibility in choosing the prior. For example, uniform priors are a possibility, which would be unnormalised for real weights. Thus, we believe that neural networks with binary weights can be an interesting alternative to the commonly used continuous ones for evaluating different components of the Bayesian deep learning pipeline.

In this section, we leverage some of the aforementioned stochastic methods for training binary neural networks to develop a variational inference algorithm with a Bernoulli prior and approximate posterior. Empirically, we find the Bernoulli prior to lead to underfitting without tempering the variational objective, so we further experiment with an ad-hoc MAP-style inference scheme under an equivalent hierarchical formulation of the prior as a Beta-Bernoulli. We evaluate the quality of the uncertainty estimates through their calibration on the test set, predictive entropy on out-of-distribution data and accuracy on a mixed dataset of test and out-of-distribution data. We find that our Bayesian approaches lead to higher uncertainty on out-of-distribution data and better accuracy on the most confident predictions than ensembles of deterministic binary networks. Finally, we show that our methods can effectively prevent forgetting in an online setting.

Statement of contributions The work in this section was carried out in collaboration with my colleague Peter Hayes and my supervisor David Barber. We developed the theoretical framework together, Peter and I collaborated on designing the code base and I carried out the experiments and wrote the text with feedback from Peter and David.

4.1.1 Binary neural networks

4.1.1.1 Deterministic binary neural networks

Neural networks are typically trained by minimising a loss $\mathcal{L}$ w.r.t. the network parameters $\boldsymbol{\theta}$

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = E(\mathcal{D}; \boldsymbol{\theta}) + \tau \mathcal{R}(\boldsymbol{\theta}), \tag{4.1}$$

where $\mathcal{L}$ is composed of an error term E (e.g. cross-entropy, squared error) that depends on the data $\mathcal{D}$ (e.g. images and class labels) and a regularisation term $\mathcal{R}$ that is only a function of the parameters and scaled by a factor τ. While for full-precision parameters a local optimum can readily be found by computing gradients via automatic differentiation [18] in one of the many modern numerical computation libraries, e.g. [269], and performing stochastic gradient descent [30], this is less straight-forward for binary parameters. Typically, binary parameters are defined to be the discretisation of some continuous parameters $\theta_i^\circ = \text{sign}(\theta_i)$, mapping $\mathbb{R}$ to $\{-1, 1\}$. This causes the gradients w.r.t. the continuous parameters to become 0, since the derivative of the sign function is 0 almost everywhere.

A simple, but surprisingly effective solution was proposed in [48, 49]: by using the 'straight-through estimator' [21], i.e. defining $\partial\theta_i^\circ/\partial\theta_i := 1$. In conjunction with small tricks like constraining the continuous weights to lie in $[-1, 1]$, binary neural networks can be trained to achieve similar accuracies as their full-precision counterparts [5].

4.1.1.2 Stochastic binary neural networks

An alternative approach is to train a distribution over the weights via variational optimisation [318, 23], i.e. to sample binary weights from a distribution $\boldsymbol{\theta}^\circ \sim q_{\boldsymbol{\phi}}(\boldsymbol{\theta})$, with parameters $\boldsymbol{\phi}$ (e.g. probabilities of a Bernoulli) and minimise the expected loss w.r.t. $\boldsymbol{\phi}$

$$\boldsymbol{\phi}^* = \arg\min_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}}\left[\mathcal{L}(\boldsymbol{\theta}^\circ)\right]. \tag{4.2}$$

Gradients can be estimated using the score-function estimator [357]. This sidesteps the non-differentiability of the sign function, although the gradients typically have high variance. One can also employ a relaxed binary distribution [224, 153] and differentiate w.r.t. continuous weights that are nearly -1 or 1, resulting in a small bias but lower variance.

Another solution specific to neural networks [309] is to notice that operations such as matrix multiplications and convolutions add up scalar products of the weights with some inputs. As long as the distribution over the weights has some variance

$\mathbb{V}[\theta_i^\circ] > 0$ and a known mean $\mathbb{E}[\theta_i^\circ]$, we can employ the central limit theorem and model the pre-activations as having a Gaussian distribution

$$\boldsymbol{\theta}^\circ \sim \prod_i q(\theta_i^\circ) \;\text{ and }\; \mathrm{h} = \sum_i \theta_i^\circ \mathrm{x}_i \;\Rightarrow\; \mathrm{h} \sim \mathcal{N}(\mu, \sigma^2), \tag{4.3}$$

with $\mu = \sum_i \mathrm{x}_i \mathbb{E}[\theta_i^\circ]$ and $\sigma^2 = \sum_i \mathrm{x}_i^2 \mathbb{V}[\theta_i^\circ]$ and $\mathbf{x}$ being the input to some layer. During training, we can then use the reparameterisation trick [167] and sample the pre-activations – rather than the weights – while maintaining differentiability w.r.t. $\boldsymbol{\phi}$.

4.1.2 Binary variational inference for neural networks

We will now briefly lay out the background for Bayesian inference before showing how stochastic optimisation of binary neural networks can be cast as variational inference. To keep the notation uncluttered, we denote all model parameters as $\boldsymbol{\theta}$ from now, as it will be either irrelevant or clear from the context if parameters are binary or continuous.

4.1.2.1 Bayesian inference

Rather than choosing some optimal parameters, Bayesian inference maintains uncertainty over all possible settings by weighing each one according to the posterior. The posterior is proportional to a prior distribution $p(\boldsymbol{\theta})$ times the likelihood of the data $p(\mathcal{D}|\boldsymbol{\theta})$, i.e. $p(\boldsymbol{\theta}|\mathcal{D}) = p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})/p(\mathcal{D})$. Predictions on new data $\mathcal{D}^*$ are made by integrating over the posterior $p(\mathcal{D}^*|\mathcal{D}) = \mathbb{E}_{p(\boldsymbol{\theta}|\mathcal{D})}[p(\mathcal{D}^*|\boldsymbol{\theta})]$.

Variational inference As calculating the evidence $p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta}$ involves an intractable integral, approximations are needed. One approach that scales well to large datasets and models is that of variational inference [130, 346]. On a high level, it turns the integration problem of $p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})d\theta$ into an optimisation problem. The aim is to find the parameters of an approximate distribution $q_{\boldsymbol{\phi}}$ that minimise the KL divergence to the true posterior. Specifically, the objective

is to maximise the evidence lower bound (ELBO)

$$\log p(\mathcal{D}) = \log \mathbb{E}_{p(\boldsymbol{\theta})}\left[p(\mathcal{D}|\boldsymbol{\theta})\right] \tag{4.4}$$

$$\geq \mathbb{E}_{q_{\boldsymbol{\phi}}}\left[\log \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{q_{\boldsymbol{\phi}}(\boldsymbol{\theta})}\right] \tag{4.5}$$

$$= \underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}}\left[\log p(\mathcal{D}|\boldsymbol{\theta})\right]}_{-E(\mathcal{D};\boldsymbol{\theta})} - \underbrace{\mathbb{KL}[q_{\boldsymbol{\phi}}\,||\,p]}_{\mathcal{R}(\boldsymbol{\theta})}, \tag{4.6}$$

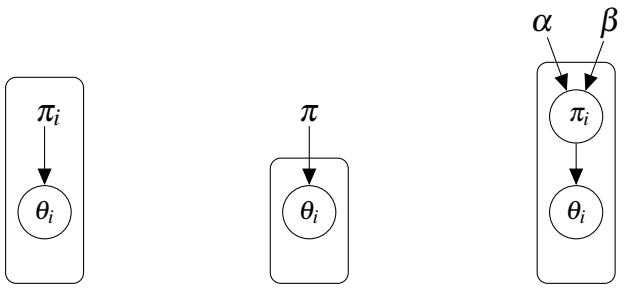
where the inequality follows from Jensen's inequality and $q_{\boldsymbol{\phi}}$ is the variational posterior that is optimised to approximate $p(\boldsymbol{\theta}|\mathcal{D})$. The first term in Eq. 4.6, the expected log likelihood, involves a sum over the data points, which can be estimated by subsampling [136]. Further, the expectation over $q_{\boldsymbol{\phi}}$ can be approximated via Monte Carlo integration [332]. Unbiased gradients can be calculated if it is possible to sample from the distribution in way that the sample is differentiable w.r.t. $\boldsymbol{\theta}$. This is the case if a sample can be obtained by transforming an independent noise variable using $\boldsymbol{\phi}$ [167]. While by default the scaling factor τ on the KL term is equal to 1, various works have found $\tau < 1$ to improve performance in the context of deep learning, e.g. [260].

Continual learning Sequentially arriving data can naturally be integrated into the Bayesian framework. When observing a new dataset $\mathcal{D}_2$ after having inferred $p(\boldsymbol{\theta}|\mathcal{D}_1)$, that posterior can be used in place of the prior since $p(\boldsymbol{\theta}|\mathcal{D}_1,\mathcal{D}_2) \propto p(\mathcal{D}_2|\boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D}_1)$ for $\mathcal{D}_1,\mathcal{D}_2$ independent given the parameters.

In an online setting, i.e. to train on a new dataset without re-using previous data, one can simply replace the prior p with the current approximate posterior $q_{\boldsymbol{\phi}}$ and train a new approximate posterior with its own parameters [259, 138, 254].

4.1.2.2 Binary Bayes-by-Backprop

To specify a variational inference algorithm for binary neural networks, we need to choose two distributions: the prior p and the approximate posterior q. We will begin with the approximate posterior as it will be the same for both choices of the prior that we consider, however with slightly differing derivations.



(a) Bernoulli inference **(b)** Bernoulli prior **(c)** Beta-Bernoulli prior

Figure 4.1: Graphical model for our Bernoulli inference distribution and the Bernoulli/Beta-Bernoulli prior over the weights $\boldsymbol{\theta}$.

Inference For inference we will assume all $\boldsymbol{\theta}$ to be independent and model each weight as Bernoulli-distributed. Strictly speaking, the Bernoulli assumes that $\theta_i \in \{0,1\}$, however we can rescale the distribution by a factor of 2 and shift it by -1 as in [271] to obtain a binary distribution over $\{-1,1\}$. The approximate posterior is then $q_{\boldsymbol{\phi}}(\boldsymbol{\theta}) = \prod_i q_{\pi_i}(\theta_i)$ with π_i denoting the probability of θ_i being 1. To estimate the expected log likelihood of the data, we can thus use the local reparameterisation trick as proposed in [309] and sample the linear outputs of convolutional and dense layers from a Gaussian distribution. From a modelling perspective, this is the binary equivalent of training a variational Gaussian posterior for continuous weights as in e.g. Bayes-by-Backprop [26] using the local reparameterisation trick [168]. See Fig. 4.1a for the graphical model.

Bernoulli Prior The first option for the prior that suggests itself is to similarly model each weight as independently Bernoulli-distributed with a common probability parameter π_{prior}. As there is no apparent reason to express a prior preference for a weight to be -1 rather than 1 or vice versa, we will use a uniform distribution with $\pi_{prior} = 0.5$. The graphical model can be found in Fig. 4.1b.

The constant uniform prior leads to the non-data dependent KL part of the ELBO being equal to the entropy $\mathbb{H}(q_{\boldsymbol{\phi}})$ of the approximate posterior plus $c := D \log \pi_{prior} = -D \log 2$

$$-\mathbb{KL}[q_{\boldsymbol{\phi}} \,||\, p] = \mathbb{E}_{q_{\boldsymbol{\phi}}}\left[\log \frac{\pi_{prior}}{q_{\boldsymbol{\phi}}(\boldsymbol{\theta})}\right] = c + \mathbb{H}(q_{\boldsymbol{\phi}}) \tag{4.7}$$

$$= c + \sum_i -\pi_i \log \pi_i - (1-\pi_i)\log(1-\pi_i). \tag{4.8}$$

The functional form of the entropy regulariser is almost identical to the probability decay term/variance regulariser in [309, 271], see Fig. 4.2. However, motivated by the aim to obtain a single well-performing binary network, they *minimise* rather than maximise the entropy/variance of $q_{\boldsymbol{\phi}}$ in contrast to the ELBO.

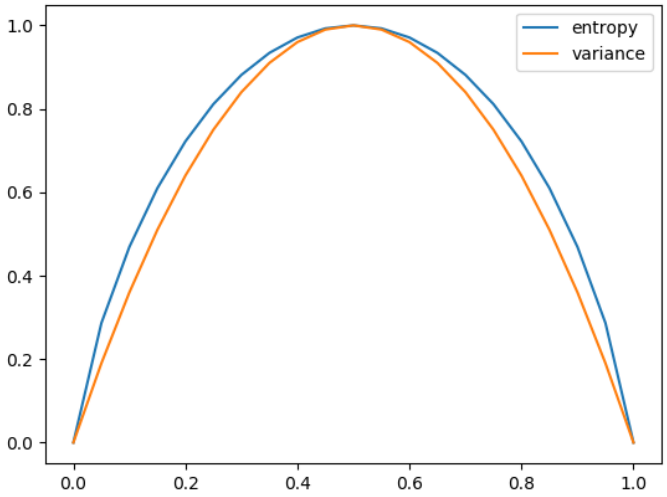


Figure 4.2: Normalised (divided by the maximum value as this can be absorbed into a multiplicative hyperparameter) entropy and variance for a Bernoulli distribution with parameter $\boldsymbol{\pi}$ (x-axis).

While the uninformative uniform prior plays no role for a single dataset, it matters when training online. The approximate posterior learns probabilities that are closer to 0 or 1 on the first dataset. On the next task the KL term regularises against shifting the mode of the weight distributions to the opposite value, allowing the network to maintain performance on previous datasets.

Beta-Bernoulli Prior A clear limitation of the Bernoulli prior is that it does not account for uncertainty over π_{prior}. As will be clear from the experiments, this prior induces overly uncertain weight distributions when not downscaling the KL term by $\tau < 1$, leading to poor predictive performance. For an alternative training procedure that allows more principled control over the regularisation of π, we consider a hierarchical extension of the Bernoulli prior, placing a prior over the probability π. We make the canonical choice of a Beta distribution as a prior over the unit interval. The graphical model is in Fig. 4.1c.

The Beta distribution has two parameters denoted as α, β. Their ratio determines the mean probability as $\mathbb{E}[\pi] = (1+\beta/\alpha)^{-1}$. As we do not express a preference for $\pi \neq 0.5$, we only consider parameterisations with $\alpha = \beta$. The Beta distribution is only normalised for $\alpha, \beta > 1$, so we denote our single parameter as $\Delta \in [0, \infty]$ and set $\alpha, \beta = \Delta + 1$.

In contrast to the Bernoulli model, the Beta prior allows us to encourage or discourage uncertain distributions over the weights. By choosing increasingly large values of Δ, we can express a prior preference for probability values that concentrate around 0.5. While in principle we could also choose $-1 < \Delta < 0$, this improper prior would lead to an unnormalised posterior, as any deterministic setting of the weights (which is the case for $\pi \in \{0, 1\}$) has infinite mass in the prior and a non-zero likelihood of the data for regression and classification. Hence, any such deterministic weight setting would have infinite mass in the posterior as well.

Note that with a Beta prior over π, we can marginalise the the probability variable out and are left with the original Bernoulli prior. We ran exploratory experiments with a Beta-Bernoulli as the variational distribution, integrating out the Beta distribution for the neural network forward pass and calculating the KL divergence as the KL between the Betas as the conditional Bernoullis cancel out. However, we were unable to achieve a good fit on the training data with these posteriors. An avenue to explore in this direction could be the collapsed variational bounds of [335].

For a more ad-hoc fix, as for the Bernoulli prior, we model the approximate

posterior over $\boldsymbol{\theta}$ as a factorised Bernoulli. Inference is then a MAP-style procedure with $\boldsymbol{\pi}^* = \arg\max_{\boldsymbol{\pi}} \log p(\mathcal{D}, \boldsymbol{\pi})$. Since we need to marginalise the weights, the resulting objective is a lower bound similar to the ELBO

$$\log p(\mathcal{D}|\boldsymbol{\pi}) + \log p_\Delta(\boldsymbol{\pi}) = \log \mathbb{E}_{p(\boldsymbol{\theta}|\boldsymbol{\pi})}\left[p(\mathcal{D}|\boldsymbol{\pi}, \boldsymbol{\theta})\right] + \log p_\Delta(\boldsymbol{\pi}) \tag{4.9}$$

$$\geq \mathbb{E}_{p(\boldsymbol{\theta}|\boldsymbol{\pi})}\left[\log p(\mathcal{D}|\boldsymbol{\theta})\right] + \log p_\Delta(\boldsymbol{\pi}). \tag{4.10}$$

Alternatively, this could be seen as variational inference with a Delta distribution on $\boldsymbol{\pi}$. This requires treating the entropy of the Delta distribution, which is infinite, as a simple constant. Probabilistic programming packages typically provide such a deterministic distribution for variational inference [22, 336] treating it as discrete. This makes the entropy zero, but implies inferring a continuous variable with a discrete one.

Modelling the inference distribution in the Beta-Bernoulli prior as a Bernoulli corresponds to placing a deterministic distribution on $\boldsymbol{\pi}$. The joint for $\boldsymbol{\pi}$ and $\boldsymbol{\theta}$ is then $q_{\boldsymbol{\pi}^*}(\boldsymbol{\pi})q(\boldsymbol{\theta}|\boldsymbol{\pi}) = \delta_{\boldsymbol{\pi}^*}(\boldsymbol{\pi})p(\boldsymbol{\theta}|\boldsymbol{\pi})$, where $\delta_{\boldsymbol{\pi}^*}(\boldsymbol{\pi})$ denotes a Delta distribution with parameter $\boldsymbol{\pi}^*$, such that $p(\boldsymbol{\pi}{=}\boldsymbol{\pi}^*) = 1$.

The entropy of the Delta distribution being constant leaves only the prior term of the KL divergence, as the conditional distributions over $\boldsymbol{\theta}$ are identical in prior and approximate posterior

$$-\mathbb{KL}[q_{\boldsymbol{\phi}} \,||\, p]) = -\mathbb{E}_{q_{\boldsymbol{\phi}}}\left[\log \frac{\delta_{\boldsymbol{\pi}^*}(\boldsymbol{\pi})}{p_\Delta(\boldsymbol{\pi})} \frac{p(\boldsymbol{\theta}^\circ|\cancel{\boldsymbol{\pi}})}{\cancel{p(\boldsymbol{\theta}^\circ|\boldsymbol{\pi})}}\right] \tag{4.11}$$

$$= \log p_\Delta(\boldsymbol{\pi}^*) + c = \Delta \sum_i \log \pi_i^*(1-\pi_i^*) + c, \tag{4.12}$$

where $c = \mathbb{H}\left[\delta_{\boldsymbol{\pi}^*}(\boldsymbol{\pi})\right]$ denotes the entropy of the Delta distribution, which is infinite when treating it as a continuous distribution and 0 for a discrete one.

While MAP-style inference may appear as a somewhat 'un-Bayesian' choice here since it neglects uncertainty over the probability values in the posterior, it allows for a side-by-side comparison of the log likelihood computations. Algorithmically,

the log likelihood of the data is estimated in the exact same way in both cases since the forward passes are the same and the only difference lies in the regularisation term. So even while under the ad-hoc MAP approach we account for the uncertainty over $\boldsymbol{\theta}$, so the *joint* approximate posterior is not a point mass, but a distribution, the objective is not a lower bound on the evidence and we cannot attribute any potential empirical success to it being Bayesian.

Algorithm 1: Forward pass for a variational binary neural network with local reparameterisation

Input: Logit parameter matrices $\boldsymbol{\phi} = [\text{vec}(\boldsymbol{W}_1), \ldots, \text{vec}(\boldsymbol{W}_L)]$
Non-linearities $f_1, \ldots, f_L$
Layer sizes $D_1, \ldots, D_L$
Data point $\mathbf{x}$
Output: Predictions for the label of $\mathbf{x}$

function BinaryForwardLR
  $\mathbf{a}_0 \leftarrow \mathbf{x}$
  for $l \in \{1, \ldots, L\}$ **do**
    $\boldsymbol{P} \leftarrow \sigma(\boldsymbol{W}_l)$
    $\boldsymbol{M} \leftarrow 2\boldsymbol{P} - 1$
    $\boldsymbol{V} \leftarrow 4\boldsymbol{P} \odot (1 - \boldsymbol{P})$
    $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}_{D_l}, \boldsymbol{I}_{D_l})$
    $\mathbf{h}_l \leftarrow \boldsymbol{M}\mathbf{a}_{l-1} + \sqrt{\boldsymbol{V}\mathbf{a}_{l-1}^2} \odot \boldsymbol{\varepsilon}$
    $\mathbf{a}_l \leftarrow f_l(\mathbf{h}_l)$
  return $\mathbf{a}_L$

We provide pseudocode for the forward pass in Alg. 1 to complement the more theoretical discussion in Section 4.1.1.2 on calculating the forward pass of a stochastic neural network in a differentiable manner. A minimal implementation for PyTorch linear layers can be found in Appendix B to illustrate that the method integrates well into modern deep learning frameworks.

4.1.3 Experiments

We run experiments on a range of vision classification tasks using a categorical log likelihood model (cross-entropy loss). We train convolutional neural networks on MNIST [194] and CIFAR10 [178]. Architectures and optimisation mostly follow [309]. One notable difference is that we also use binary weights in the output layer.

For this we rescale the linear outputs of the network (the logits) by the inverse square root of the size of the last hidden layer. This is to limit the variance of the logits to 1.[1]

On MNIST, we use the following architecture

$$\text{Conv}(64) - \text{MP}(2) - \text{Tanh} - \text{Conv}(128) - \text{MP}(2) - \text{Tanh} - \text{Linear}(2048) - \text{Tanh} - \text{Linear}(10)$$

where $\text{Conv}(c)$ denotes a 3×3 2d convolutional layer with c channels, stride 1 and padding $\lfloor \frac{c}{2} \rfloor$. $\text{MP}(s)$ denotes 2d max-pooling over an $s \times s$ patch with stride s. $\text{Linear}(d)$ is a linear layer with output-dimensionality d.

On CIFAR10, our architecture is

$$\begin{aligned}&\text{Conv}(128) - \text{BN} - \text{Tanh} - \text{Conv}(128) - \text{MP}(2) - \text{BN} - \text{Tanh} - \\ &\text{Conv}(256) - \text{BN} - \text{Tanh} - \text{Conv}(256) - \text{MP}(2) - \text{BN} - \text{Tanh} - \\ &\text{Conv}(512) - \text{BN} - \text{Tanh} - \text{Conv}(512) - \text{MP}(2) - \text{BN} - \text{Tanh} - \\ &\text{Linear}(1024) - \text{BN} - \text{Tanh} - \text{Linear}(10),\end{aligned}$$

where BN refers to a batch normalisation layer.

We found multiplying the pre-activations in each layer (prior to the non-linearity) by the inverse square root of the number of inputs to slightly improve accuracy on MNIST for the binary networks, so we use this rescaling throughout all layers. On CIFAR10, results were almost identical, so we only rescale the linear outputs. This rescaling could be seen as inferring binary weights of value $\{-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}}\}$ rather than $\{-1, 1\}$, where d is the dimensionality of the previous layer. The motivation would be, similar to full precision BNNs, to preserve variance in the forward pass by using a $\mathcal{N}(0, \frac{1}{d})$ prior on the weights [251].

Similar to Shayer et al. [309] we found initialising the continuous parameters

[1] The variance of the weight distribution and that of the inputs to the last layer are both at most 1 as we use tanh non-linearities. The variance of each logit value is therefore bounded by the number of terms.

of the binary networks to the values of the weights of a full precision network with the same architecture to slightly improve performance. Note that we did not transfer the parameters or statistics of the batch normalisation layers. We do not use Dropout [317] in either architecture as this is a popular tool of its own for estimating the uncertainty in neural networks [83], although it does not match the performance of ensembles [185].

We train parameters using the Adam optimiser [166] with default hyperparameters using PyTorch [269] and decay the learning by a factor of 0.1. For initial learning rate and further hyperparameters, see Tab. 4.1. We do not use any data augmentation on MNIST and randomly flip and take crops of size 32×32 after padding by 4 pixels on all sides on CIFAR10. For the deterministic networks, we clip the real-valued parameters to lie in $[-1, 1]$.

We compare to deterministic binary neural networks trained with the straight-through estimator (STE) and ensembles of such binary neural networks. Ensembles have been shown to provide reliable uncertainty estimates for full precision neural networks [185, 315]. We repeat our experiments 8 times (ensembles are repeated 4 times) and plots display the mean of the respective quantities with two standard errors as the shaded area around them. We use 8 samples from the approximate posterior for prediction and ensembles of the same size.

Table 4.1: Learning rate, epochs and decay schedule for straight-through estimator and Binary Bayes-by-Backprop.

	MNIST		CIFAR10		pMNIST	
	STE	BBbB	STE	BBbB	STE	BBbB
Initial learning rate	0.1	0.1	0.1	0.01	0.01	0.01
Total epochs	200	200	300	300	200	200
Decay epochs	80, 160	25, 125	120, 240	200	100	100

4.1.3.1 Accuracy and log likelihoods

We report test accuracy and negative log likelihood for MNIST and CIFAR10 in Tab. 4.2 for a selection of prior parameters and scaling factors on the KL term. More extensive results, also including Brier scores [34], for 4 and 16 posterior samples are in Tabs. D.2 and D.3 in the appendix. On both datasets, ensembles perform

Table 4.2: MNIST/CIFAR10 accuracies and log likelihoods. Both the ensemble size and the number of samples from the approximate posterior are equal to 8.

	MNIST		CIFAR10	
	NLL	Accuracy (%)	NLL	Accuracy (%)
STE	0.027 ± 0.001	99.21 ± 0.02	0.371 ± 0.004	92.84 ± 0.06
Ensemble (STE)	0.022 ± 0.0	99.25 ± 0.01	0.189 ± 0.002	94.32 ± 0.13
Bernoulli (τ=1)	0.237 ± 0.001	95.21 ± 0.05	1.108 ± 0.013	64.9 ± 0.52
Bernoulli (τ=10^{-2})	0.033 ± 0.0	98.99 ± 0.02	0.318 ± 0.002	90.35 ± 0.11
Bernoulli (τ=10^{-3})	0.032 ± 0.001	98.97 ± 0.02	0.27 ± 0.002	91.73 ± 0.06
Beta-Bernoulli (Δ=0)	0.029 ± 0.0	99.02 ± 0.01	0.275 ± 0.002	91.91 ± 0.1
Beta-Bernoulli (Δ=10^{-3})	0.028 ± 0.0	99.03 ± 0.03	0.284 ± 0.001	91.16 ± 0.09
Beta-Bernoulli (Δ=10^{-4})	0.028 ± 0.0	99.06 ± 0.02	0.271 ± 0.001	91.84 ± 0.05

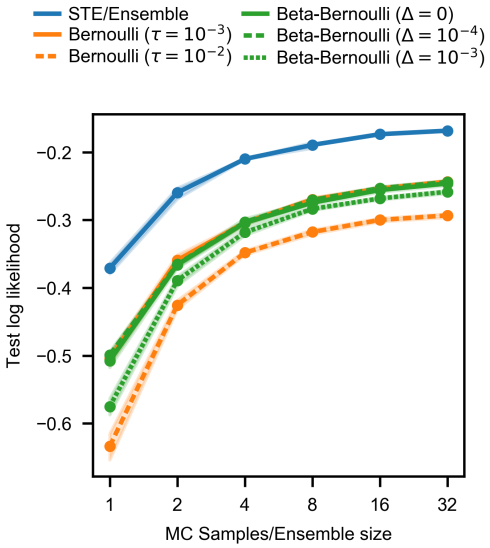
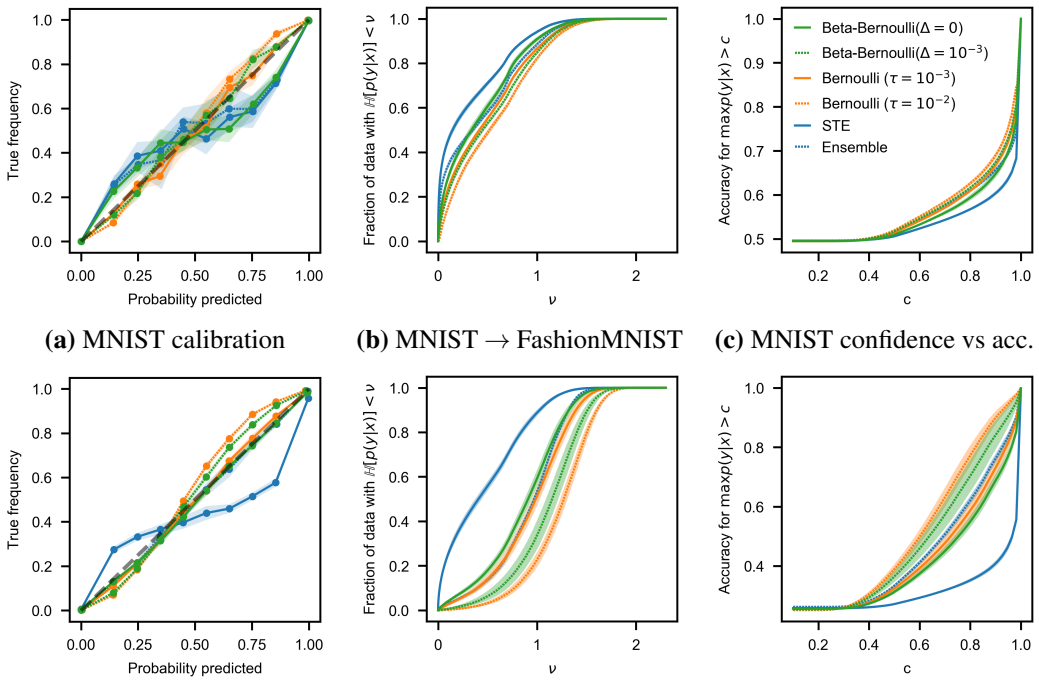


Figure 4.3: CIFAR10 test log likelihood as a function of ensemble size/number of samples from the approximate posterior.

the best in terms of accuracy and likelihood, although the gap is rather small on MNIST. While the single deterministic network has slightly higher accuracy than the Bayesian ones, most of the latter have a lower negative log likelihood when not encouraging overly uncertain distributions. As one would expect, setting a prior with a higher Δ, i.e. forcing $\boldsymbol{\pi}$ to be closer to 0.5 inhibits classification performance. The Bernoulli prior leads to severe underfitting, especially on CIFAR10, unless one significantly downweights the KL term in the ELBO by setting τ<1. This failure highlights a fundamental issue with the Bernoulli model, namely that specifying indifference regarding the value of the weight is insufficient, and that a preference



(a) MNIST calibration **(b)** MNIST → FashionMNIST **(c)** MNIST confidence vs acc.

(d) CIFAR10 calibration **(e)** CIFAR10 → SVHN **(f)** CIFAR10 confidence vs acc.

Figure 4.4: Uncertainty evaluation on MNIST (top) and CIFAR10 (bottom). Left column shows calibration on the test set (closer to diagonal is better), middle column shows the fraction of OOD data (FashionMNIST and SVHN respectively) for which the entropy of the prediction is below ν (lower is better), and right column displays the accuracy of predictions with the maximum predicted probability above c on the merged test and OOD dataset (data from the latter always count as mis-classified).

over how peaked the distribution is is needed. We think that in light of these results, hierarchical priors for continuous BNNs as used e.g. in [123] could be worth revisiting.

In Fig. 4.3 we show the log likelihood on CIFAR10 as a function of the number of samples/ensemble size. We observe a greater performance improvement from drawing additional samples for methods that encourage stronger weight uncertainty (larger Δ, τ), although the base performance for a single sample is typically lower.

4.1.3.2 Uncertainty evaluation

Beyond basic classification accuracy, we also investigate the quality of the uncertainty estimates. We show calibration curves (probability predicted for a class vs. the corresponding empirical frequency) in Figs. 4.4a and 4.4d. For both networks, we find the predictions made using deterministic weights to be overconfident. En-

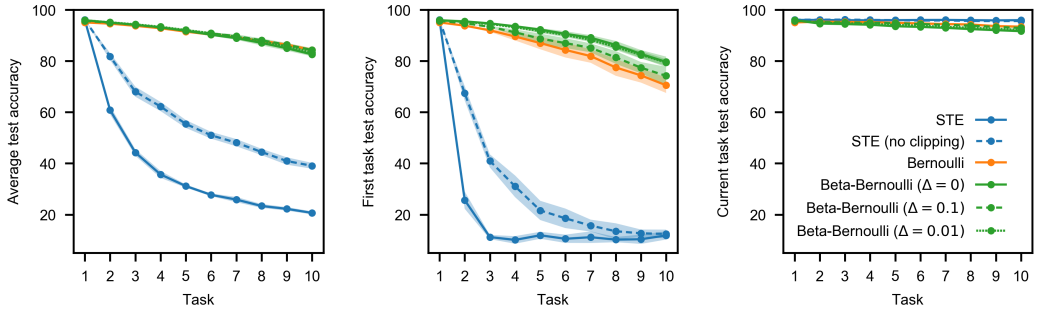


Figure 4.5: Continual learning on permuted MNIST. Left subfigure shows the average test accuracy across all tasks the network has been trained on, central and right subfigures show the test accuracy on the first task (ability to remember) and the most current task (flexibility on new data) respectively.

sembles can remedy this problem on CIFAR10, but the predictions of the different networks are too similar on MNIST to make a difference. The Bayesian networks are generally well-calibrated on MNIST when encouraging weight uncertainty, but slightly underconfident on CIFAR10, although a uniform prior on $\boldsymbol{\pi}$ leads to good calibration.

Further, we display cumulative densities of the predictive entropies ($\mathbb{H}(p) = -\sum_i p(y_i|x) \log p(y_i|x)$) on out-of-distribution (OOD) data in Figs. 4.4b and 4.4e. For MNIST, we use FashionMNIST [363] as OOD data and SVHN [253] for CIFAR10. As expected, the deterministic network is significantly more certain than the alternatives. The Bayesian variants exhibit higher uncertainty on OOD data than ensembles when encouraging uncertainty on the weights ($\Delta > 0$ for the Beta-Bernoulli prior). Combining the corresponding test and OOD sets, we show the accuracy of predictions above a moving confidence threshold c in Figs. 4.4c and 4.4f similar to Lakshminarayanan et al. [185]. Again, the deterministic networks fare the worst, with the Bayesian methods being more accurate than the ensembles for uncertainty-encouraging priors. Notably, approximating the predictive posterior with a single sample leads to *over*confidence of the Bayesian networks on CIFAR10, however they still outperform the deterministic one. 2 samples lead to substantial improvement and most of the benefit of ensembling can be achieved with 4 samples.

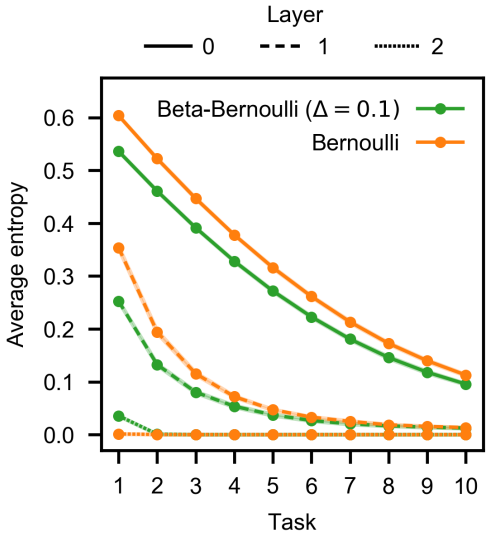


Figure 4.6: Mean entropy of the weight distributions per layer on permuted MNIST.

4.1.3.3 Continual learning

Finally, we train a fully connected network with two hidden layers of 100 units on a sequence of ten instantiations of permuted MNIST [95]. While this is a somewhat simple problem, it offers the advantage of providing a sequence of problems that are of equal size and difficulty (since fully connected networks are invariant to a permutation of their inputs).

We plot the classification accuracy throughout training in Fig. 4.5. We compare sequentially training a deterministic binary network to using a Bernoulli and Beta-Bernoulli prior. For the latter, we use three different parameters ($\Delta \in \{0, 0.1, 0.01\}$). We also train a deterministic network without clipping the continuous weights.

The left subplot shows the average test accuracy across datasets the network has been trained on so far, the central subplot the accuracy on the very first task as a measure of the network's ability to remember old data, and the subplot on the right performance on the current task to measure flexibility to adapt to new data, as the parameters may be over-regularised toward old values.

As expected, the deterministic network forgets how to perform previous tasks immediately. Interestingly, not clipping the weights slows down forgetting without reducing the accuracy on the current task. This provides support for the hypothesis

that continuous 'latent' weights provide inertia during training [119]. The Bayesian networks only see a slow and gradual performance decay on old data. Between them, the difference in the average performance is negligible, which is to be expected as from the second task onward they all model the weights as Bernoulli-distributed in the prior. We do, however observe that the Beta-Bernoulli prior leads to better memorisation of the first task. This is due to the prior encouraging less uncertain weight distributions as displayed in Fig. 4.6 using the average per-weight entropy as a measure of uncertainty.

4.1.4 Related work

There are three notable prior works on Bayesian binary neural networks. Soudry et al. [316] propose an online algorithm based on expectation propagation [238] for binary targets. They report better performance for their binary variant over the Gaussian one. Su et al. [319] propose a similar deterministic method based on variational inference, but require significant downweighting of the KL term to obtain good performance. Most recently, Meng et al. [235] introduce a binary instantiation of the Bayesian learning rule [163]. Their main focus lies on justifying heuristics used in the training of deterministic binary neural networks from a Bayesian viewpoint, specifically the use of the straight-through estimator [48, 49] and exponential moving averages of gradients for determining sign flips [119]. However, they mainly focus on classification accuracies and only evaluate uncertainty estimates on toy data.

There is a rapidly growing literature on full precision BNNs. Most works formulate their prior model and perform inference in weights space, aiming to develop flexible approximate posteriors that can account for correlations while remaining computationally efficient. We would position the method in this section as the binary equivalent of [26], i.e. 'Binary Bayes-by-Backprop'. As in [26] we perform stochastic variational inference by subsampling the data and estimating the log likelihood with Monte Carlo samples from the approximate posterior and learn the parameters of the corresponding distribution using automatic differentiation and gradient descent. Both works use the standard distributions for the respective domain of the approximate posterior – the Bernoulli over a binary random variable

in our case and the Gaussian for real numbers in the case of [26]. We use the local reparameterisation trick to sample the pre-activations rather than the weights to maintain differentiability w.r.t. the parameters. This marginalisation is exact in the case of Gaussian parameters [168], but an approximation for binary ones — albeit an accurate one as empirically demonstrated in [309].

4.1.5 Conclusion

This section proposed a method for variational inference as well as a closely related ad-hoc approach for binary neural networks that leverage recent progress on training stochastic binary neural networks. We investigated a uniform prior on the weights as well as a hierarchical variation and found that the hierarchical prior allows for a more probabilistically motivated formulation of an effective training objective. While ensembles of binary neural networks offer stronger test log likelihood and classification accuracy, we found the uncertainty estimates resulting from the two methods developed in this section to be more robust to out-of-distribution data. Future directions for this work include an extension to ternary weights, which offer additional flexibility and avenue towards sparsifying the networks. Further, distributions that account for uncertainty over the probability parameter in the approximate posterior, in particular structured ones, may achieve better performance thanks to their more expressive posteriors.

4.2 Sparse uncertainty representation with inducing weights

Deep learning models are becoming deeper and wider than ever before. From image recognition models such as ResNet-101 [114, 115] and DenseNet [143] to BERT [364] and GPT-3 [36] for language modelling, deep neural networks have found consistent success in fitting large-scale data. As these models are increasingly deployed in real-world applications, calibrated uncertainty estimates for their predictions become crucial, especially in safety-critical areas such as healthcare. In this regard, BNNs [222, 26, 83, 374] and deep ensembles [185] represent two popular paradigms for estimating uncertainty, which have shown promising results in applications such as (medical) image processing [161, 326] and out-of-distribution detection [315].

Though progress has been made, one major obstacle to scaling up BNNs and deep ensembles is the high cost in storing them. Both approaches require the parameter counts to be several times higher than their deterministic counterparts. Although recent efforts have improved memory efficiency [215, 324, 353, 62], they still require storage memory that is higher than storing a deterministic neural network.

Meanwhile, an *infinitely wide* BNN becomes a Gaussian process (GP) that is known for good uncertainty estimates [250, 230, 196]. But perhaps surprisingly, this infinitely-wide BNN is "parameter-efficient", as its "parameters" are effectively the datapoints, which have a considerably smaller memory footprint than explicitly storing the network weights. To further reduce the computational burden, sparse posterior approximations with a small number of *inducing points* are widely used [314, 331], rendering sparse GPs more memory efficient than their neural network counterparts.

Can we bring the advantages of sparse approximations in GPs – which are infinitely-wide neural networks – to finite width deep learning models? We provide an affirmative answer regarding memory efficiency, by proposing an uncertainty quantification framework based on *sparse uncertainty representations*. We present our approach in the BNN context, but the proposed approach is also applicable to deep ensembles. In detail, our contributions in this section are as follows:

- We introduce *inducing weights* — an auxiliary variable method with lower dimensional counterparts to the actual weight matrices — for variational inference in BNNs, as well as a memory efficient parameterisation and an extension to ensemble methods (Section 4.2.2).

- We extend Matheron's rule to facilitate efficient posterior sampling (Section 4.2.2.2).

- We provide an in-depth computation complexity analysis (Section 4.2.3), showing the significant advantage in terms of parameter-efficiency.

- We show the connection to sparse (deep) GPs, in that inducing weights can be viewed as *projected noisy inducing outputs* in pre-activation output space (Section 4.2.6).

- We apply the proposed approach to BNNs and deep ensembles. Experiments in classification, model robustness and out-of-distribution detection tasks show that our inducing weight approaches achieve competitive performance to their counterparts in the original weight space on modern deep architectures for image classification, while reducing the parameter count to $\leq 24.3\%$ of that of a single network.

Statement of contributions The work in this section was carried out in collaboration with Martin Kukla, Cheng Zhang and Yingzhen Li during an internship at Microsoft Research Cambridge. Yingzhen and I developed the theoretical framework with feedback from Cheng, I designed the codebase with feedback from Martin and Yingzhen. Yingzhen and I ran the experiments and wrote the paper [290] together with feedback from Martin and Cheng. The function space perspective in Section 4.2.6 was developed and written by Yingzhen and is included here from the paper for completeness. An implementation of the method proposed in this section alongside utility code for converting regular PyTorch neural networks into (variationally) Bayesian ones is available at `https://github.com/Microsoft/bayesianize`.

4.2.1 Inducing variables for variational inference

Our work is built on variational inference and inducing variables for posterior approximations. Given observations $\mathcal{D} = \{\mathbf{X}, \mathbf{Y}\}$ with $\mathbf{X} = [\boldsymbol{x}_1, ..., \boldsymbol{x}_N]$, $\mathbf{Y} = [\boldsymbol{y}_1, ..., \boldsymbol{y}_N]$, we would like to fit a neural network $p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{W}_{1:L})$ with weights $\boldsymbol{W}_{1:L}$ to the data. BNNs posit a prior distribution $p(\boldsymbol{W}_{1:L})$ over the weights, and construct an approximate posterior $q(\boldsymbol{W}_{1:L})$ to the intractable exact posterior $p(\boldsymbol{W}_{1:L}|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{W}_{1:L})p(\boldsymbol{W}_{1:L})$, where $p(\mathcal{D}|\boldsymbol{W}_{1:L}) = p(\mathbf{Y}|\mathbf{X}, \boldsymbol{W}_{1:L}) = \prod_{n=1}^{N} p(\boldsymbol{y}_n|\boldsymbol{x}_n, \boldsymbol{W}_{1:L})$.

Variational inference Variational inference [155, 372] constructs an approximation $q(\boldsymbol{\theta})$ to the posterior $p(\boldsymbol{\theta}|\mathcal{D}) \propto p(\boldsymbol{\theta})p(\mathcal{D}|\boldsymbol{\theta})$ by maximising a variational lower-bound

$$\log p(\mathcal{D}) \geq \text{ELBO}(q(\boldsymbol{\theta})) := \mathbb{E}_{q(\boldsymbol{\theta})}\left[\log p(\mathcal{D}|\boldsymbol{\theta})\right] - \mathbb{KL}[q(\boldsymbol{\theta}) \,||\, p(\boldsymbol{\theta})]. \tag{4.13}$$

For BNNs, $\boldsymbol{\theta} = \{\boldsymbol{W}_{1:L}\}$, and a simple choice of q is a fully factorised Gaussian (FFG): $q(\boldsymbol{W}_{1:L}) = \prod_{l=1}^{L}\prod_{i=1}^{d^l_{out}}\prod_{j=1}^{d^l_{in}} \mathcal{N}(M^l_{i,j}, V^l_{i,j})$, with $M^l_{i,j}, V^l_{i,j}$ the mean and variance of $W^l_{i,j}$ and d^l_{in}, d^l_{out} the respective number of inputs and outputs to layer l. The variational parameters are then $\boldsymbol{\phi} = \{\boldsymbol{M}_l, \boldsymbol{V}_l\}_{l=1}^{L}$. Gradients of ELBO w.r.t. $\boldsymbol{\phi}$ can be estimated with mini-batches of data [136] and with Monte Carlo sampling from the q distribution [332, 167]. By setting q to an FFG, a variational BNN can be trained with similar computational requirements as a deterministic network [26].

Improved posterior approximation with inducing variables Auxiliary variable approaches [3, 299, 279] construct the $q(\boldsymbol{\theta})$ distribution with an auxiliary variable $\mathbf{a}$: $q(\boldsymbol{\theta}) = \int q(\boldsymbol{\theta}|\mathbf{a})q(\mathbf{a})d\mathbf{a}$, with the hope that a potentially richer mixture distribution $q(\boldsymbol{\theta})$ can achieve better approximations. As then $q(\boldsymbol{\theta})$ becomes intractable, an auxiliary variational lower-bound is used to optimise $q(\boldsymbol{\theta}, \mathbf{a})$

$$\log p(\mathcal{D}) \geq \text{ELBO}(q(\boldsymbol{\theta}, \mathbf{a})) = \mathbb{E}_{q(\boldsymbol{\theta}, \mathbf{a})}[\log p(\mathcal{D}|\boldsymbol{\theta})] + \mathbb{E}_{q(\boldsymbol{\theta}, \mathbf{a})}\left[\log \frac{p(\boldsymbol{\theta})r(\mathbf{a}|\boldsymbol{\theta})}{q(\boldsymbol{\theta}|\mathbf{a})q(\mathbf{a})}\right]. \tag{4.14}$$

Here $r(\mathbf{a}|\boldsymbol{\theta})$ is an auxiliary distribution that needs to be specified, where existing approaches often use a "reverse model" for $r(\mathbf{a}|\boldsymbol{\theta})$. Instead, we define $r(\mathbf{a}|\boldsymbol{\theta})$ in a generative manner: $r(\mathbf{a}|\boldsymbol{\theta})$ is the "posterior" of the following "generative model",

whose "evidence" is exactly the prior of $\boldsymbol{\theta}$

$$r(\mathbf{a}|\boldsymbol{\theta}) = \tilde{p}(\mathbf{a}|\boldsymbol{\theta}) \propto \tilde{p}(\mathbf{a})\tilde{p}(\boldsymbol{\theta}|\mathbf{a}), \quad \text{such that } \tilde{p}(\boldsymbol{\theta}) := \int \tilde{p}(\mathbf{a})\tilde{p}(\boldsymbol{\theta}|\mathbf{a})d\mathbf{a} = p(\boldsymbol{\theta}). \tag{4.15}$$

Plugging Eq. 4.15 into Eq. 4.14 yields

$$\text{ELBO}(q(\boldsymbol{\theta},\mathbf{a})) = \mathbb{E}_{q(\boldsymbol{\theta})}[\log p(\mathcal{D}|\boldsymbol{\theta})] - \mathbb{E}_{q(\mathbf{a})}\left[\mathbb{KL}[q(\boldsymbol{\theta}|\mathbf{a}) \,||\, \tilde{p}(\boldsymbol{\theta}|\mathbf{a})]\right] - \mathbb{KL}[q(\mathbf{a}) \,||\, \tilde{p}(\mathbf{a})]. \tag{4.16}$$

This approach returns an efficient approximate inference algorithm, translating the complexity of inference in $\boldsymbol{\theta}$ to $\mathbf{a}$, if $\dim(\mathbf{a}) < \dim(\boldsymbol{\theta})$ and $q(\boldsymbol{\theta},\mathbf{a}) = q(\boldsymbol{\theta}|\mathbf{a})q(\mathbf{a})$ has the following properties:

1. A "pseudo prior" $\tilde{p}(\mathbf{a})\tilde{p}(\boldsymbol{\theta}|\mathbf{a})$ is defined such that $\int \tilde{p}(\mathbf{a})\tilde{p}(\boldsymbol{\theta}|\mathbf{a})d\mathbf{a} = p(\boldsymbol{\theta})$;
2. The conditionals $q(\boldsymbol{\theta}|\mathbf{a})$ and $\tilde{p}(\boldsymbol{\theta}|\mathbf{a})$ are in the same parametric family, so can share parameters;
3. Both sampling $\boldsymbol{\theta} \sim q(\boldsymbol{\theta})$ and computing $\mathbb{KL}[q(\boldsymbol{\theta}|\mathbf{a}) \,||\, \tilde{p}(\boldsymbol{\theta}|\mathbf{a})]$ can be done efficiently;
4. The designs of $q(\mathbf{a})$ and $\tilde{p}(\mathbf{a})$ can potentially provide extra advantages (in time and space complexities and/or optimisation easiness).

We call $\mathbf{a}$ the *inducing variable* of $\boldsymbol{\theta}$, which is inspired by variationally sparse GP (SVGP) with inducing points [314, 331]. Indeed SVGP is a special case: $\boldsymbol{\theta} = \mathbf{f}$, $\mathbf{a} = \mathbf{u}$, the GP prior is $p(\mathbf{f}|\mathbf{X}) = \mathcal{GP}(\mathbf{0}, \mathbf{K_{XX}})$, $p(\mathbf{u}) = \mathcal{GP}(\mathbf{0}, \mathbf{K_{ZZ}})$, $\tilde{p}(\mathbf{f},\mathbf{u}) = p(\mathbf{u})p(\mathbf{f}|\mathbf{X},\mathbf{u})$, $q(\mathbf{f}|\mathbf{u}) = p(\mathbf{f}|\mathbf{X},\mathbf{u})$, $q(\mathbf{f},\mathbf{u}) = p(\mathbf{f}|\mathbf{X},\mathbf{u})q(\mathbf{u})$, and $\mathbf{Z}$ are the optimisable inducing inputs. The variational lower-bound is $\text{ELBO}(q(\mathbf{f},\mathbf{u})) = \mathbb{E}_{q(\mathbf{f})}[\log p(\mathbf{Y}|\mathbf{f})] - \mathbb{KL}[q(\mathbf{u}) \,||\, p(\mathbf{u})]$, and the variational parameters are $\boldsymbol{\phi} = \{\mathbf{Z}, \text{distribution parameters of } q(\mathbf{u})\}$. SVGP satisfies the marginalisation constraint Eq. 4.15 by definition, and it has $\mathbb{KL}[q(\mathbf{f}|\mathbf{u}) \,||\, \tilde{p}(\mathbf{f}|\mathbf{u})] = 0$. Also by using small $M = \dim(\mathbf{u})$ and exploiting the q distribution design, SVGP reduces run-time from $\mathcal{O}(N^3)$ to $\mathcal{O}(NM^2)$ where N is the number of inputs in $\mathbf{X}$, meanwhile

it also makes storing a full Gaussian $q(\mathbf{u})$ affordable. Lastly, $\mathbf{u}$ can be whitened, leading to the "pseudo prior" $\tilde{p}(\mathbf{f},\mathbf{v}) = p(\mathbf{f}|\mathbf{X},\mathbf{u}=\mathbf{K}_{\mathbf{ZZ}}^{1/2}\mathbf{v})\tilde{p}(\mathbf{v}), \tilde{p}(\mathbf{v}) = \mathcal{N}(\mathbf{v};\mathbf{0},\boldsymbol{I})$ which could bring potential benefits in optimisation.

We emphasise that the introduction of 'pseudo prior' does *not* change the *probabilistic model* as long as the marginalisation constraint Eq. 4.15 is satisfied. In the rest of the section we assume the constraint Eq. 4.15 holds and write $p(\boldsymbol{\theta},\mathbf{a}) := \tilde{p}(\boldsymbol{\theta},\mathbf{a})$. It might seem unclear how to design such $\tilde{p}(\boldsymbol{\theta},\mathbf{a})$ for an arbitrary probabilistic model, however, for a Gaussian prior on $\boldsymbol{\theta}$ the rules for computing conditional Gaussian distributions can be used to construct $\tilde{p}$. In Section 4.2.2 we exploit these rules to develop an efficient approximate inference method for BNNs with inducing weights.

4.2.2 Sparse uncertainty representation with inducing weights

4.2.2.1 Inducing weights for neural network parameters

Following the above design principles, we introduce to each network layer l a *smaller* inducing weight matrix $\boldsymbol{U}_l$ to assist approximate posterior inference in $\boldsymbol{W}_l$. Therefore in our context, $\boldsymbol{\theta} = \boldsymbol{W}_{1:L}$ and $\mathbf{a} = \boldsymbol{U}_{1:L}$. In the rest of the section, we assume a factorised prior across layers $p(\boldsymbol{W}_{1:L}) = \prod_l p(\boldsymbol{W}_l)$, and drop the l indices when the context is clear to ease notation.

Augmenting network layers with inducing weights Suppose the weight $\boldsymbol{W} \in \mathbb{R}^{d_{out}\times d_{in}}$ has a Gaussian prior $p(\boldsymbol{W}) = p(\text{vec}(\boldsymbol{W})) = \mathcal{N}(\mathbf{0},\sigma^2\boldsymbol{I})$ where $\text{vec}(\boldsymbol{W})$ concatenates the columns of the weight matrix into a vector and σ is the standard deviation. A first attempt to augment $p(\text{vec}(\boldsymbol{W}))$ with an inducing weight variable $\boldsymbol{U} \in \mathbb{R}^{M_{out}\times M_{in}}$ may be to construct a multivariate Gaussian $p(\text{vec}(\boldsymbol{W}),\text{vec}(\boldsymbol{U}))$, such that $\int p(\text{vec}(\boldsymbol{W}),\text{vec}(\boldsymbol{U}))d\boldsymbol{U} = \mathcal{N}(\mathbf{0},\sigma^2\boldsymbol{I})$.

This means that the block corresponding to the covariance of $\text{vec}(\boldsymbol{W})$ in the joint covariance matrix of $(\text{vec}(\boldsymbol{W}),\text{vec}(\boldsymbol{U}))$ needs to match the prior covariance $\sigma^2\boldsymbol{I}$. We are then free to parameterise the remaining of the entries in the joint covariance matrix, as long as this full matrix remains positive definite. Augmenting this Gaussian with an auxiliary variable $\boldsymbol{U}$ that also has a mean of zero and some

covariance that we are free to parameterise, the joint distribution can be written as

$$\begin{pmatrix} \mathrm{vec}(\boldsymbol{W}) \\ \mathrm{vec}(\boldsymbol{U}) \end{pmatrix} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}) \quad \text{with} \quad L = \begin{pmatrix} \sigma \boldsymbol{I} & \mathbf{0} \\ \boldsymbol{Z} & \boldsymbol{D} \end{pmatrix}$$
$$\text{s.t.} \quad \boldsymbol{\Sigma} = LL^\top = \begin{pmatrix} \sigma^2 \boldsymbol{I} & \sigma \boldsymbol{Z}^\top \\ \sigma \boldsymbol{Z} & \boldsymbol{Z}\boldsymbol{Z}^\top + \boldsymbol{D}^2 \end{pmatrix},$$

where $\boldsymbol{D}$ is a positive diagonal matrix and $\boldsymbol{Z}$ a matrix with arbitrary entries. Through defining the Cholesky decomposition of $\boldsymbol{\Sigma}$ we ensure its positive definiteness. By the usual rules of Gaussian marginalisation, the augmented model leaves the marginal prior on $\boldsymbol{W}$ unchanged. Further, we can analytically derive the conditional distribution on the weights given the inducing weights

$$p(\mathrm{vec}(\boldsymbol{W}) | \mathrm{vec}(\boldsymbol{U})) = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{W}|\boldsymbol{U}}, \boldsymbol{\Sigma}_{\boldsymbol{W}|\boldsymbol{U}}), \tag{4.17}$$
$$\boldsymbol{\mu}_{\boldsymbol{W}|\boldsymbol{U}} = \sigma \boldsymbol{Z}^\top \boldsymbol{\Psi}^{-1} \mathrm{vec}(\boldsymbol{U}), \tag{4.18}$$
$$\boldsymbol{\Sigma}_{\boldsymbol{W}|\boldsymbol{U}} = \sigma^2 (\boldsymbol{I} - \boldsymbol{Z}^\top \boldsymbol{\Psi}^{-1} \boldsymbol{Z}), \tag{4.19}$$
$$\boldsymbol{\Psi} = \boldsymbol{Z}\boldsymbol{Z}^\top + \boldsymbol{D}^2.$$

For inference, we now need to define an approximate posterior over the joint space $q(\boldsymbol{W}, \boldsymbol{U})$. We will do so by factorising it as $q(\boldsymbol{W}, \boldsymbol{U}) = q(\boldsymbol{W}|\boldsymbol{U})q(\boldsymbol{U})$. Factorising the prior in the same way leads to the following KL term in the ELBO

$$\mathbb{KL}[q(\boldsymbol{W}, \boldsymbol{U}) \,||\, p(\boldsymbol{W}, \boldsymbol{U})] = \mathbb{E}_{q(\boldsymbol{U})} \left[\mathbb{KL}[q(\boldsymbol{W}|\boldsymbol{U}) \,||\, p(\boldsymbol{W}|\boldsymbol{U}]] + \mathbb{KL}[q(\boldsymbol{U}) \,||\, p(\boldsymbol{U})] \right] \tag{4.20}$$

Matrix normal augmentation Unfortunately, as $\dim(\mathrm{vec}(\boldsymbol{W}))$ is typically large (e.g. of the order of 10^7), using a full covariance Gaussian for $p(\mathrm{vec}(\boldsymbol{W}), \mathrm{vec}(\boldsymbol{U}))$ becomes computationally intractable.

We address this issue with matrix normal distributions [107]. The prior $p(\mathrm{vec}(\boldsymbol{W})) = \mathcal{N}(\mathbf{0}, \sigma^2 \boldsymbol{I})$ has an equivalent matrix normal distribution form as $p(\boldsymbol{W}) = \mathcal{MN}(0, \sigma_r^2 \boldsymbol{I}, \sigma_c^2 \boldsymbol{I})$, with $\sigma_r, \sigma_c > 0$ the row and column standard deviations satisfying $\sigma = \sigma_r \sigma_c$.

Now we introduce the inducing variable $\boldsymbol{U}$ in matrix space, as well as two auxiliary variables $\boldsymbol{U}_r \in \mathbb{R}^{M_{out} \times d_{in}}$, $\boldsymbol{U}_c \in \mathbb{R}^{d_{out} \times M_{in}}$, so that the full augmented prior is

$$\begin{pmatrix} \boldsymbol{W} & \boldsymbol{U}_c \\ \boldsymbol{U}_r & \boldsymbol{U} \end{pmatrix} \sim p(\boldsymbol{W}, \boldsymbol{U}_c, \boldsymbol{U}_r, \boldsymbol{U}) := \mathcal{MN}(0, \boldsymbol{\Sigma}_r, \boldsymbol{\Sigma}_c), \tag{4.21}$$

$$\text{with} \quad \boldsymbol{L}_r = \begin{pmatrix} \sigma_r \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{Z}_r & \boldsymbol{D}_r \end{pmatrix} \quad \text{s.t.} \quad \boldsymbol{\Sigma}_r = \boldsymbol{L}_r \boldsymbol{L}_r^\top = \begin{pmatrix} \sigma_r^2 \boldsymbol{I} & \sigma_r \boldsymbol{Z}_r^\top \\ \sigma_r \boldsymbol{Z}_r & \boldsymbol{Z}_r \boldsymbol{Z}_r^\top + \boldsymbol{D}_r^2 \end{pmatrix}$$

$$\text{and} \quad \boldsymbol{L}_c = \begin{pmatrix} \sigma_c \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{Z}_c & \boldsymbol{D}_c \end{pmatrix} \quad \text{s.t.} \quad \boldsymbol{\Sigma}_c = \boldsymbol{L}_c \boldsymbol{L}_c^\top = \begin{pmatrix} \sigma_c^2 \boldsymbol{I} & \sigma_c \boldsymbol{Z}_c^\top \\ \sigma_c \boldsymbol{Z}_c & \boldsymbol{Z}_c \boldsymbol{Z}_c^\top + \boldsymbol{D}_c^2 \end{pmatrix}.$$

See Fig. 4.7(a) for a visualisation. The marginalisation constraint Eq. 4.15 is satisfied for any $\boldsymbol{Z}_c \in \mathbb{R}^{M_{in} \times d_{in}}, \boldsymbol{Z}_r \in \mathbb{R}^{M_{out} \times d_{out}}$ and diagonal matrices $\boldsymbol{D}_c, \boldsymbol{D}_r$. The marginal distribution of $\boldsymbol{U}$ is $p(\boldsymbol{U}) = \mathcal{MN}(\boldsymbol{0}, \boldsymbol{\Psi}_r, \boldsymbol{\Psi}_c)$ with $\boldsymbol{\Psi}_r = \boldsymbol{Z}_r \boldsymbol{Z}_r^\top + \boldsymbol{D}_r^2$ and $\boldsymbol{\Psi}_c = \boldsymbol{Z}_c \boldsymbol{Z}_c^\top + \boldsymbol{D}_c^2$. In the experiments we use *whitened* inducing weights which transforms $\boldsymbol{U}$ so that $p(\boldsymbol{U}) = \mathcal{MN}(\boldsymbol{0}, \boldsymbol{I}, \boldsymbol{I})$ (Section 4.2.2.3), but for clarity we continue with the above formulas.

Matrix normal distributions have similar marginalisation and conditioning properties as multivariate Gaussian distributions. The marginal both over some set of rows and some set of columns is still a matrix normal. Hence, $p(\boldsymbol{W}) = \mathcal{MN}(\boldsymbol{0}, \sigma_r^2 \boldsymbol{I}, \sigma_c^2 \boldsymbol{I})$, and by choosing $\sigma_r \sigma_c = \sigma$ this matrix normal distribution is equivalent to the multivariate normal $p(\text{vec}(\boldsymbol{W})) = \mathcal{N}(\boldsymbol{0}, \sigma^2 \boldsymbol{I})$. Also $p(\boldsymbol{U}) = \mathcal{MN}(\boldsymbol{0}, \boldsymbol{\Psi}_r, \boldsymbol{\Psi}_c)$, where again $\boldsymbol{\Psi}_r = \boldsymbol{Z}_r \boldsymbol{Z}_r^\top + \boldsymbol{D}_r^2$ and $\boldsymbol{\Psi}_c = \boldsymbol{Z}_c \boldsymbol{Z}_c^\top + \boldsymbol{D}_c^2$. Similarly, the conditionals on some rows

or columns are matrix normal distributed

$$
\begin{aligned}
\boldsymbol{U}_c|\boldsymbol{U} &\sim \mathcal{MN}(\sigma_r \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1}\boldsymbol{U}, \sigma_r^2(\boldsymbol{I}-\boldsymbol{Z}_r^\top\boldsymbol{\Psi}_r^{-1}\boldsymbol{Z}_r), \boldsymbol{\Psi}_c),\\
\boldsymbol{U}_r|\boldsymbol{U} &\sim \mathcal{MN}(\boldsymbol{U}\boldsymbol{\Psi}_c^{-1}\sigma_c\boldsymbol{Z}_c, \boldsymbol{\Psi}_r, \sigma_c^2(\boldsymbol{I}-\boldsymbol{Z}_c^\top\boldsymbol{\Psi}_c^{-1}\boldsymbol{Z}_c)),\\
\boldsymbol{W}|\boldsymbol{U}_c &\sim \mathcal{MN}\left(\boldsymbol{U}_c\boldsymbol{\Psi}_c^{-1}\sigma_c\boldsymbol{Z}_c, \sigma_r^2\boldsymbol{I}, \sigma_c^2(\boldsymbol{I}-\boldsymbol{Z}_c^\top\boldsymbol{\Psi}_c^{-1}\boldsymbol{Z}_c)\right),\\
\boldsymbol{W},\boldsymbol{U}_r|\boldsymbol{U}_c,\boldsymbol{U} &\sim \mathcal{MN}(\begin{pmatrix}\boldsymbol{U}_c\\ \boldsymbol{U}\end{pmatrix}\boldsymbol{\Psi}_c^{-1}\sigma_c\boldsymbol{Z}_c, \boldsymbol{\Sigma}_r, \sigma_c^2(\boldsymbol{I}-\boldsymbol{Z}_c^\top\boldsymbol{\Psi}_c^{-1}\boldsymbol{Z}_c)),\\
\boldsymbol{W}|\boldsymbol{U}_r,\boldsymbol{U}_c,\boldsymbol{U} &\sim \mathcal{MN}(M_{\boldsymbol{W}}, \sigma_r^2(\boldsymbol{I}-\boldsymbol{Z}_r^\top\boldsymbol{\Psi}^{-1}\boldsymbol{Z}_r), \sigma_c^2(\boldsymbol{I}-\boldsymbol{Z}_c^\top\boldsymbol{\Psi}_c^{-1}\boldsymbol{Z}_c)),\\
M_{\boldsymbol{W}} &= \sigma(\boldsymbol{Z}_r^\top\boldsymbol{\Psi}_r^{-1}\boldsymbol{U}_r + \boldsymbol{U}_c\boldsymbol{\Psi}_c^{-1}\boldsymbol{Z}_c - \boldsymbol{Z}_r^\top\boldsymbol{\Psi}_r^{-1}\boldsymbol{U}\boldsymbol{\Psi}_c^{-1}\boldsymbol{Z}_c).
\end{aligned}
\tag{4.22}
$$

The matrix normal parameterisation introduces two additional variables $\boldsymbol{U}_r, \boldsymbol{U}_c$ without providing additional expressiveness. Hence it is desirable to integrate them out, leading to a joint multivariate normal with Khatri-Rao product structure for the covariance

$$
p(\mathrm{vec}(\boldsymbol{W}), \mathrm{vec}(\boldsymbol{U})) = \mathcal{N}\left(\boldsymbol{0}, \begin{pmatrix} \sigma_c^2\boldsymbol{I}\otimes\sigma_r^2\boldsymbol{I} & \sigma_c\boldsymbol{Z}_c^\top\otimes\sigma_r\boldsymbol{Z}_r^\top \\ \sigma_c\boldsymbol{Z}_c\otimes\sigma_r\boldsymbol{Z}_r & \boldsymbol{\Psi}_c\otimes\boldsymbol{\Psi}_r \end{pmatrix}\right). \tag{4.23}
$$

As the dominating memory complexity here is $\mathcal{O}(d_{out}M_{out} + d_{in}M_{in})$ which comes from storing $\boldsymbol{Z}_r$ and $\boldsymbol{Z}_c$, we see that the matrix normal parameterisation of the augmented prior is memory efficient.

Posterior approximation in the joint space We construct a factorised posterior approximation across the layers: $q(\boldsymbol{W}_{1:L}, \boldsymbol{U}_{1:L}) = \prod_l q(\boldsymbol{W}_l|\boldsymbol{U}_l)q(\boldsymbol{U}_l)$. Below we discuss options for $q(\boldsymbol{W}|\boldsymbol{U})$. For the conditional distribution on the weights, in the simplest case we set $q(\boldsymbol{W}|\boldsymbol{U}) = p(\boldsymbol{W}|\boldsymbol{U}) = p(\mathrm{vec}(\boldsymbol{W})|\,\mathrm{vec}(\boldsymbol{U})) = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{W}|\boldsymbol{U}}, \boldsymbol{\Sigma}_{\boldsymbol{W}|\boldsymbol{U}})$, hence the KL divergence would be zero similar to sparse GPs.

For the most general case of arbitrary Gaussian distributions with $q = \mathcal{N}(\boldsymbol{\mu}_q, \boldsymbol{\Sigma}_q)$

and $p = \mathcal{N}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)$, the KL divergence is

$$\mathbb{KL}[q\,||\,p] = \frac{1}{2}(\log \frac{\det \boldsymbol{\Sigma}_p}{\det \boldsymbol{\Sigma}_q} - d + \mathrm{tr}(\boldsymbol{\Sigma}_p^{-1}\boldsymbol{\Sigma}_q) + (\boldsymbol{\mu}_p - \boldsymbol{\mu}_q)^\top \boldsymbol{\Sigma}_p^{-1}(\boldsymbol{\mu}_p - \boldsymbol{\mu}_q)), \quad (4.24)$$

where d is the number of elements of $\boldsymbol{\mu}$.

As motivated, it is desirable to make $q(\boldsymbol{W}|\boldsymbol{U})$ similar to $p(\boldsymbol{W}|\boldsymbol{U})$. Hence we consider a slightly more flexible scalar rescaling of the covariance which multiplies a term λ^2 onto the covariance matrix

$$q(\boldsymbol{W}|\boldsymbol{U}) = q(\mathrm{vec}(\boldsymbol{W})|\,\mathrm{vec}(\boldsymbol{U})) = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{W}|\boldsymbol{U}}, \lambda^2 \boldsymbol{\Sigma}_{\boldsymbol{W}|\boldsymbol{U}}). \quad (4.25)$$

This allows for efficiently computing the KL as the final term, the Mahalanobis distance between the means under p, cancels out entirely and the log determinant and trace terms become a function of λ only

$$\begin{aligned}
\mathbb{KL}[q\,||\,p] &= \frac{1}{2}(\log \frac{\det \boldsymbol{\Sigma}}{\det \lambda^2 \boldsymbol{\Sigma}} - d + \mathrm{tr}(\boldsymbol{\Sigma}^{-1}\lambda^2\boldsymbol{\Sigma})) && (4.26)\\
&= \frac{1}{2}(\log \frac{\det \boldsymbol{\Sigma}}{\lambda^{2d} \det \boldsymbol{\Sigma}} - d + \mathrm{tr}(\lambda^2 \boldsymbol{I})) && (4.27)\\
&= \frac{1}{2}(-2d\log\lambda - d + d\lambda^2) && (4.28)\\
&= d(\frac{1}{2}\lambda^2 - \log\lambda - \frac{1}{2}) =: R(\lambda), && (4.29)
\end{aligned}$$

with $d = \dim(\mathrm{vec}(\boldsymbol{W}))$.

Plugging $\boldsymbol{\theta} = \boldsymbol{W}_{1:L}, \mathbf{a} = \boldsymbol{U}_{1:L}$ and Eq. 4.29 into Eq. 4.16 results in the following variational lower-bound

$$\begin{aligned}\text{ELBO}(q(\boldsymbol{W}_{1:L}, \boldsymbol{U}_{1:L})) =& \mathbb{E}_{q(\boldsymbol{W}_{1:L})}[\log p(\mathcal{D}|\boldsymbol{W}_{1:L})] - \sum_{l=1}^{L}(R(\lambda_l) \\ &+ \mathbb{KL}[q(\boldsymbol{U}_l) \,||\, p(\boldsymbol{U}_l)]),\end{aligned} \tag{4.30}$$

with λ_l the associated scaling parameter for $q(\boldsymbol{W}_l|\boldsymbol{U}_l)$. Again as the choices of $\boldsymbol{Z}_c, \boldsymbol{Z}_r, \boldsymbol{D}_c, \boldsymbol{D}_r$ do not change the marginal prior $p(\boldsymbol{W})$, we are safe to optimise them as well. Therefore the variational parameters are now $\boldsymbol{\phi} = \{\boldsymbol{Z}_c, \boldsymbol{Z}_r, \boldsymbol{D}_c, \boldsymbol{D}_r, \lambda, \text{dist. params. of } q(\boldsymbol{U})\}$ for each layer.

Two choices of $q(\boldsymbol{U})$ A simple choice is a fully factorised Gaussian (FFG) $q(\text{vec}(\boldsymbol{U})) = \mathcal{N}(\boldsymbol{m}_u, \text{diag}(\boldsymbol{v}_u))$, which performs mean-field inference in $\boldsymbol{U}$ space [c.f. 26], and here $\mathbb{KL}[q(\boldsymbol{U}) \,||\, p(\boldsymbol{U})]$ has a closed-form solution. Another choice is a "mixture of delta measures" $q(\boldsymbol{U}) = \frac{1}{K}\sum_{k=1}^{K} \delta(\boldsymbol{U} = \boldsymbol{U}^{(k)})$, i.e. we keep K distinct sets of parameters $\{\boldsymbol{U}_{1:L}^{(k)}\}_{k=1}^{K}$ in inducing space that are projected back into the original parameter space via the *shared* conditionals $q(\boldsymbol{W}_l|\boldsymbol{U}_l)$ to obtain the weights. This approach can be viewed as constructing "deep ensembles" in $\boldsymbol{U}$ space, and we follow ensemble methods [e.g. 185] to drop $\mathbb{KL}[q(\boldsymbol{U}) \,||\, p(\boldsymbol{U})]$ in Eq. 4.30.

Often $\boldsymbol{U}$ is chosen to have significantly lower dimensions than $\boldsymbol{W}$, i.e. $M_{in} \ll d_{in}$ and $M_{out} \ll d_{out}$. As $q(\boldsymbol{W}|\boldsymbol{U})$ and $p(\boldsymbol{W}|\boldsymbol{U})$ only differ in the covariance scaling constant, $\boldsymbol{U}$ can be regarded as a *sparse representation of uncertainty* for the network layer, as the major updates in (approximate) posterior belief is quantified by $q(\boldsymbol{U})$.

4.2.2.2 Efficient sampling with the extended Matheron's rule

Computing the variational lower-bound Eq. 4.30 requires samples from $q(\boldsymbol{W})$, which requires an efficient sampling procedure for $q(\boldsymbol{W}|\boldsymbol{U})$. Unfortunately, $q(\boldsymbol{W}|\boldsymbol{U})$ derived from Eq. 4.23 & Eq. 4.25 is not a matrix normal, so *direct* sampling is prohibitive.

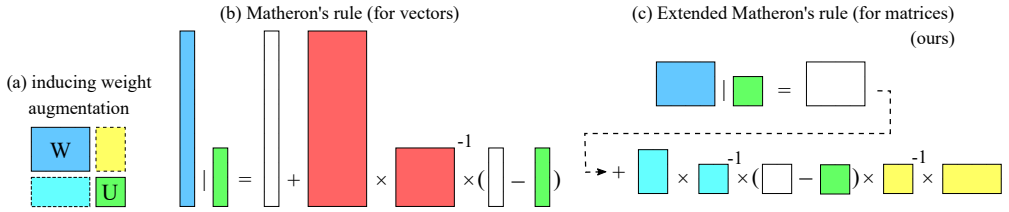


Figure 4.7: Visualisation of (a) the inducing weight augmentation, and compare (b) the original Matheron's rule to (c) our extended version. White blocks represent samples from the joint Gaussian.

To address this challenge, we extend Matheron's rule [156, 137, 59] to efficiently sample from $q(\boldsymbol{W}|\boldsymbol{U})$. As we show in Appendix C, we can sample from a conditional Gaussian $\boldsymbol{W} \sim q(\boldsymbol{W}|\boldsymbol{U})$ by transforming a sample from a joint matrix normal distribution

$$\boldsymbol{W} = \lambda\bar{\boldsymbol{W}} + \sigma\boldsymbol{Z}_r^\top\boldsymbol{\Psi}_r^{-1}(\boldsymbol{U} - \lambda\bar{\boldsymbol{U}})\boldsymbol{\Psi}_c^{-1}\boldsymbol{Z}_c; \tag{4.31}$$

$$\bar{\boldsymbol{W}}, \bar{\boldsymbol{U}} \sim p(\bar{\boldsymbol{W}}, \bar{\boldsymbol{U}}_c, \bar{\boldsymbol{U}}_r, \bar{\boldsymbol{U}}) = \mathcal{MN}(\boldsymbol{0}, \boldsymbol{\Sigma}_r, \boldsymbol{\Sigma}_c). \tag{4.32}$$

Here $\bar{\boldsymbol{W}}, \bar{\boldsymbol{U}} \sim p(\bar{\boldsymbol{W}}, \bar{\boldsymbol{U}}_c, \bar{\boldsymbol{U}}_r, \bar{\boldsymbol{U}})$ means we first sample $\bar{\boldsymbol{W}}, \bar{\boldsymbol{U}}_c, \bar{\boldsymbol{U}}_r, \bar{\boldsymbol{U}}$ from the joint then drop $\bar{\boldsymbol{U}}_c, \bar{\boldsymbol{U}}_r$; in fact $\bar{\boldsymbol{U}}_c, \bar{\boldsymbol{U}}_r$ are never computed.

Further, $\bar{\boldsymbol{W}}, \bar{\boldsymbol{U}}$ can be obtained by

$$\begin{aligned} &\bar{\boldsymbol{W}} = \sigma E_1, \ \bar{\boldsymbol{U}} = \boldsymbol{Z}_r E_1 \boldsymbol{Z}_c^\top + \hat{\boldsymbol{Z}}_r \tilde{E}_2 \boldsymbol{D}_c + \boldsymbol{D}_r \tilde{E}_3 \hat{\boldsymbol{Z}}_c^\top + \boldsymbol{D}_r E_4 \boldsymbol{D}_c, \\ &E_1 \sim \mathcal{MN}(\boldsymbol{0}, \boldsymbol{I}_{d_{out}}, \boldsymbol{I}_{d_{in}}); \quad \tilde{E}_2, \tilde{E}_3, E_4 \sim \mathcal{MN}(\boldsymbol{0}, \boldsymbol{I}_{M_{out}}, \boldsymbol{I}_{M_{in}}), \\ &\hat{\boldsymbol{L}}_r = \mathrm{chol}(\boldsymbol{Z}_r \boldsymbol{Z}_r^\top), \ \hat{\boldsymbol{L}}_c = \mathrm{chol}(\boldsymbol{Z}_c \boldsymbol{Z}_c^\top). \end{aligned} \tag{4.33}$$

The run-time cost is $\mathcal{O}(2M_{out}^3 + 2M_{in}^3 + d_{out}M_{out}M_{in} + M_{in}d_{out}d_{in})$ required by inverting $\boldsymbol{\Psi}_r, \boldsymbol{\Psi}_c$, computing $\hat{\boldsymbol{L}}_r$, $\hat{\boldsymbol{L}}_c$, and the matrix products. The extended Matheron's rule is visualised in Fig. 4.7 with a comparison to the original Matheron's rule for sampling from $q(\mathrm{vec}(\boldsymbol{W})|\,\mathrm{vec}(\boldsymbol{U}))$. Note that the original rule requires joint sampling from Eq. 4.23 (i.e. sampling the white blocks in Fig. 4.7(b)) which has $\mathcal{O}((d_{out}d_{in} + M_{out}M_{in})^3)$ cost. Therefore our recipe avoids inverting and multiplying big matrices, resulting in conditional sampling becoming computationally feasible

for neural network-sized matrices.

4.2.2.3 Whitening and hierarchical inducing variables

The inducing weights $\boldsymbol{U}_{1:L}$ further allow for introducing a single inducing weight matrix $\boldsymbol{U}$ that is shared across the network. By doing so, correlations of weights between layers in the approximate posterior are introduced. The inducing weights are then sampled jointly conditioned on the global inducing weights. This requires that all inducing weight matrices are of the same size along at least one axis, such that they can be concatenated along the other one.

The easiest way of introducing a global inducing weight matrix is by proceeding similarly to the introduction of the per-layer inducing weights. As a pre-requisite, we need to work with "whitened" inducing weights, i.e. set the covariance of the marginal $p(\boldsymbol{U}_l)$ to the identity and pre-multiply the covariance block between $\boldsymbol{W}_l$ and $\boldsymbol{U}_l$ with the inverse Cholesky of $\boldsymbol{\Psi}_l$. In this whitened model, the full augmented prior per-layer is

$$\begin{pmatrix} \boldsymbol{W} & \boldsymbol{U}_c \\ \boldsymbol{U}_r & \boldsymbol{U} \end{pmatrix} \sim p(\boldsymbol{W}, \boldsymbol{U}_c, \boldsymbol{U}_r, \boldsymbol{U}) := \mathcal{MN}(0, \tilde{\boldsymbol{\Sigma}}_r, \tilde{\boldsymbol{\Sigma}}_r), \tag{4.34}$$

$$\text{with} \quad \tilde{\boldsymbol{L}}_r = \begin{pmatrix} \sigma_r \boldsymbol{I} & 0 \\ \boldsymbol{L}_r^{-1}\boldsymbol{Z}_r & \boldsymbol{L}_r^{-1}\boldsymbol{D}_r \end{pmatrix} \text{ s.t. } \quad \tilde{\boldsymbol{\Sigma}}_r = \tilde{\boldsymbol{L}}_r \tilde{\boldsymbol{L}}_r^\top = \begin{pmatrix} \sigma_r^2 \boldsymbol{I} & \sigma_r \boldsymbol{Z}_r^\top \boldsymbol{L}_r^{-\top} \\ \sigma_r \boldsymbol{L}_r^{-1}\boldsymbol{Z}_r & \boldsymbol{I} \end{pmatrix}$$

$$\text{and} \quad \tilde{\boldsymbol{L}}_c = \begin{pmatrix} \sigma_c \boldsymbol{I} & 0 \\ \boldsymbol{L}_c^{-1}\boldsymbol{Z}_c & \boldsymbol{L}_c^{-1}\boldsymbol{D}_c \end{pmatrix} \text{ s.t. } \quad \tilde{\boldsymbol{\Sigma}}_c = \tilde{\boldsymbol{L}}_c \tilde{\boldsymbol{L}}_c^\top = \begin{pmatrix} \sigma_c^2 \boldsymbol{I} & \sigma_c \boldsymbol{Z}_c^\top \boldsymbol{L}_c^{-\top} \\ \sigma_c \boldsymbol{L}_c^{-1}\boldsymbol{Z}_c & \boldsymbol{I} \end{pmatrix}.$$

One can verify that this whitened model leads to the same conditional distribution $p(\boldsymbol{W}|\boldsymbol{U})$ as presented in the main text. After whitening, for each $\boldsymbol{U}_l$ we have that $p(\text{vec}(\boldsymbol{U}_l)) = \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$, therefore we can also write their joint distribution as $p(\text{vec}(\boldsymbol{U}_{1:L})) = \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$. In order to construct a matrix normal prior $p(\boldsymbol{U}_{1:L}) = \mathcal{MN}(0, \boldsymbol{I}, \boldsymbol{I})$, the inducing weight matrices $\boldsymbol{U}_{1:L}$ needs to be stacked either along the rows or columns, requiring the other dimension to be matching across all layers. Then, as the covariance is the identity with $\sigma = \sigma_r = \sigma_c = 1$, we can augment $p(\boldsymbol{U}_{1:L})$ in the exact same way as we previously augmented the prior $p(\boldsymbol{W}_l)$ with

Table 4.3: Computational complexity per layer. We assume $\boldsymbol{W} \in \mathbb{R}^{d_{out} \times d_{in}}$, $\boldsymbol{U} \in \mathbb{R}^{M_{out} \times M_{in}}$, and K forward passes for each of the N inputs. (* uses a parallel computing friendly vectorisation technique [353] for further speed-up.)

Method	Time complexity	Storage complexity
Deterministic-W	$\mathcal{O}(Nd_{in}d_{out})$	$\mathcal{O}(d_{in}d_{out})$
FFG-W	$\mathcal{O}(NKd_{in}d_{out})$	$\mathcal{O}(2d_{in}d_{out})$
Ensemble-W	$\mathcal{O}(NKd_{in}d_{out})$	$\mathcal{O}(Kd_{in}d_{out})$
Matrix-normal-W	$\mathcal{O}(NKd_{in}d_{out})$	$\mathcal{O}(d_{in}d_{out} + d_{in} + d_{out})$
k-tied FFG-W	$\mathcal{O}(NKd_{in}d_{out})$	$\mathcal{O}(d_{in}d_{out} + k(d_{in} + d_{out}))$
rank-1 BNN	$\mathcal{O}(NKd_{in}d_{out})^*$	$\mathcal{O}(d_{in}d_{out} + 2(d_{in} + d_{out}))$
FFG-U	$\mathcal{O}(NKd_{in}d_{out} + 2M_{in}^3 + 2M_{out}^3 + K(d_{out}M_{out}M_{in} + M_{in}d_{out}d_{in}))$	$\mathcal{O}(d_{in}M_{in} + d_{out}M_{out} + 2M_{in}M_{out})$
Ensemble-U	same as above	$\mathcal{O}(d_{in}M_{in} + d_{out}M_{out} + KM_{in}M_{out})$

$\boldsymbol{U}_l$. In preliminary experiments, we did not observe any performance improvements under the hierarchical model, so will leave this approach for future work.

4.2.3 Computational complexities

In Tab. 4.3 we report the computational complexity figures for two types of inducing weight approaches: FFG $q(\boldsymbol{U})$ (FFG-U) and Delta mixture $q(\boldsymbol{U})$ (Ensemble-U). Baseline approaches include: Deterministic-W, variational inference with FFG $q(\boldsymbol{W})$ [FFG-W, 26], deep ensemble in $\boldsymbol{W}$ [Ensemble-W, 185], as well as parameter-efficient approaches such as matrix-normal $q(\boldsymbol{W})$ (Matrix-normal-W, Louizos and Welling [215]), variational inference with k-tied FFG $q(\boldsymbol{W})$ (k-tied FFG-W, Swiatkowski et al. [324]), and rank-1 BNN [62]. The gain in memory is significant for the inducing weight approaches, in fact with $M_{in} < d_{in}$ and $M_{out} < d_{out}$ the parameter storage requirement is smaller than a single deterministic neural network. The major overhead in run-time comes from the extended Matheron's rule for sampling $q(\boldsymbol{W}|\boldsymbol{U})$. Some of the computations there are performed only once, and in our experiments we show that by using a relatively low-dimensional $\boldsymbol{U}$ and large batch-sizes, the overhead is acceptable.

4.2.4 Experiments

We evaluate the inducing weight approaches on regression, classification and related uncertainty estimation tasks. The goal is to demonstrate competitive performance to popular $\boldsymbol{W}$-space uncertainty estimation methods while using significantly fewer

parameters. Existing parameter-efficient approaches for uncertainty estimation (e.g. k-tied or rank-1 BNNs) have achieved close performance to deep ensembles. However, *none* of them reduces the parameter count to be smaller than that of a *single* network. Therefore we decide *not* to include these baselines and instead focus on comparing: (1) variational inference with FFG $q(\boldsymbol{W})$ [FFG-W, 26] v.s. FFG $q(\boldsymbol{U})$ (FFG-U, ours); (2) deep ensemble in $\boldsymbol{W}$ space [Ensemble-W, 185] v.s. ensemble in $\boldsymbol{U}$ space (Ensemble-U, ours). Another baseline is training a deterministic neural network with maximum likelihood.

The number of parameters for FFG-U and Ensemble-U with an ensemble size of 5 in the ResNet-50 experiments are reported in Tab. 4.4.

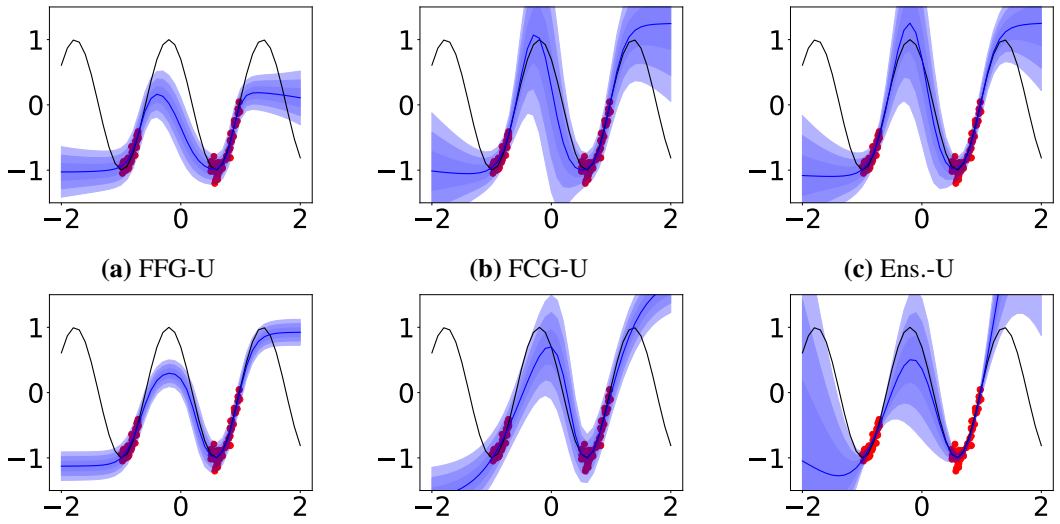
Table 4.4: Parameter counts for the inducing models with varying $\boldsymbol{U}$ size M.

M	Method	$M=16$	$M=32$	$M=64$	$M=128$	$M=256$	Deterministic
Abs. value	FFG-U	1,384,662	2,771,446	5,710,902	12,253,366	27,992,502	23,520,842
	Ensemble-U	1,426,134	2,937,334	6,374,454	14,907,574	38,609,334	
rel. size (%)	FFG-U	5.89	11.78	24.28	52.10	119.01	100
	Ensemble-U	6.06	12.49	27.10	63.38	164.15	

4.2.4.1 Synthetic 1-D regression

The synthetic regression task follows Foong et al. [74], which has two clusters of inputs $x_1 \sim \mathcal{U}[-1,-0.7]$, $x_2 \sim \mathcal{U}[0.5,1]$, and targets $y \sim \mathcal{N}(\cos(4x+0.8),0.01)$. For reference we show the exact posterior results using the NUTS sampler [135]. The results are visualised in Fig. 4.8 with predictive mean in blue, and up to three standard deviations as shaded area. Similar to historical results, FFG-W fails to represent the increased uncertainty away from the data and in between clusters. While underestimating predictive uncertainty overall, FFG-U shows a small increase in predictive uncertainty away from the data. In contrast, a per-layer full covariance Gaussian in both weight (FCG-W) and inducing space (FCG-U) as well as Ensemble-U better capture the increased predictive variance, although the mean function is more similar to that of FFG-W.

Following [74], we sample 50 inputs each from $\mathcal{U}[-1,-0.7]$ and $\mathcal{U}[0.5,1]$ as inputs and targets $y \sim \mathcal{N}(\cos(0.4x+0.8),0.01)$. As a prior we use a zero-mean Gaussian with standard deviation $\frac{4}{\sqrt{d_{in}}}$ for the weights and biases of each layer. Our



(a) FFG-U **(b)** FCG-U **(c)** Ens.-U

(d) FFG-W **(e)** FCG-W **(f)** NUTS

Figure 4.8: Toy regression results, with observations in red dots and the ground truth function in black.

network architecture has a single hidden layer of 50 units and uses a tanh non-linearity. All three variational methods are optimised using Adam [166] for $20,000$ updates with an initial learning rate of 10^{-3}. We average over 32 MC samples from the approximate posterior for every update. For Ensemble-U and FCG-U we decay the learning rate by a factor of 0.1 after $10,000$ updates and the size of the inducing weight matrix is 2×25 for the input layer (accounting for the bias) and 25×1 for the output layer. Ensemble-U uses an ensemble size of 8.

For NUTS we use the implementation provided in Pyro [22]. We draw a total of $25,000$ samples, discarding the first 5000 as burn-in and using 1000 randomly selected ones for prediction.

4.2.4.2 Classification and in-distribution calibration

Table 4.5: Complete in-distribution results for Resnet-50 on CIFAR10

Method	Acc. ↑	NLL ↓	ECE ↓	Brier ↓
Deterministic	94.72±0.08	0.43±0.01	4.46±0.08	0.10±0.00
Ensemble-W	95.90	0.20	1.08	0.06
FFG-W	94.13±0.08	0.18±0.00	0.50±0.06	0.09±0.00
FFG-U (M=64)	94.40±0.05	0.17±0.00	0.64±0.06	0.08±0.00
FFG-U (M=128)	94.66±0.09	0.17±0.00	1.59±0.06	0.08±0.00
Ensemble-U (M=64)	94.94±0.07	0.16±0.00	0.45±0.06	0.08±0.00
Ensemble-U (M=128)	95.34±0.05	0.17±0.00	1.29±0.05	0.07±0.00

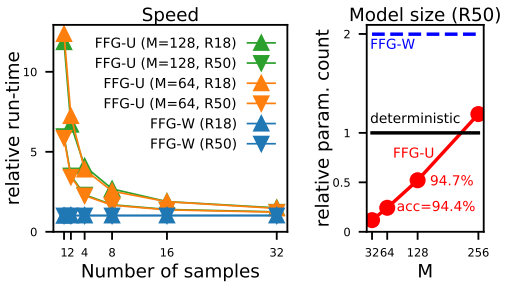


Figure 4.9: Resnet run-times & model sizes.

Table 4.6: Complete in-distribution results for Resnet-50 on CIFAR100

Method	Acc. ↑	NLL ↓	ECE ↓	Brier ↓
Deterministic	75.73±0.16	2.14±0.01	19.69±0.15	0.43±0.00
Ensemble-W	79.33	1.23	6.51	0.31
FFG-W	74.44±0.27	1.01±0.01	4.24±0.10	0.35±0.00
FFG-U (M=64)	75.37±0.09	0.92±0.01	2.29±0.39	0.34±0.00
FFG-U (M=128)	75.88±0.13	0.91±0.00	6.66±0.15	0.34±0.00
Ensemble-U (M=64)	75.97±0.12	0.90±0.00	1.12±0.06	0.33±0.00
Ensemble-U (M=128)	77.61±0.11	0.94±0.00	6.00±0.12	0.32±0.00

As the core empirical evaluation, we train Resnet-50 models [115] on CIFAR-10 and CIFAR-100 [179]. To avoid underfitting issues with FFG-W, a useful trick is to set an upper limit σ^2_{max} on the variance of $q(\boldsymbol{W})$ [215]. This trick is similarly applied to the $\boldsymbol{U}$-space methods, where we cap $\lambda \leq \lambda_{max}$ for $q(\boldsymbol{W}|\boldsymbol{U})$, and for FFG-U we also set σ^2_{max} for the variance of $q(\boldsymbol{U})$. In convolution layers, we treat the 4D weight tensor $\boldsymbol{W}$ of shape (c_{out}, c_{in}, h, w) as a $c_{out} \times c_{in}hw$ matrix. We use $\boldsymbol{U}$ matrices of shape 64×64 for all layers (i.e. $M = M_{in} = M_{out} = 64$), except that for CIFAR-10 we set $M_{out} = 10$ for the last layer.

In Tabs. 4.5 and 4.6 we report test accuracy and test ECE [106] along NLLs and Brier scores as a evaluation of the uncertainty estimates. This table contains two standard errors across the random seeds for the corresponding metrics. The error bar results are not available for Ensemble-W, as it is constructed from the 5 independently trained deterministic neural network with maximum likelihood.

Overall, Ensemble-W achieves the highest accuracy, but is not as well-calibrated as variational methods. For the inducing weight approaches, Ensemble-U outperforms FFG-U on both datasets; overall it performs the best on the more challenging

CIFAR-100 dataset (close-to-Ensemble-W accuracy and lowest ECE). Tabs. 4.5 and 4.6 in the Appendix show that increasing the $\boldsymbol{U}$ dimensions to $M = 128$ improves accuracy but leads to slightly worse calibration.

In Fig. 4.9 we show prediction run-times for batch-size $= 500$ on an NVIDIA Tesla V100 GPU, relative to those of an ensemble of deterministic nets, as well as relative parameter sizes to a single ResNet-50. The extra run-times for the inducing methods come from computing the extended Matheron's rule. However, as they can be calculated once and cached for drawing multiple samples, the overhead reduces to a small factor when using larger number of samples K, especially for the bigger Resnet-50. More importantly, when compared to a *deterministic* ResNet-50, the inducing weight models reduce the parameter count by over 75% ($5,710,902$ vs. $23,520,842$) for $M = 64$.

4.2.4.3 Ablation study

We visualise in Fig. 4.10 the accuracy and ECE results for computationally lighter inducing weight ResNet-18 models with different hyper-parameters. Performance in both metrics improves as the $\boldsymbol{U}$ matrix size M is increased (right-most panels), and the results for $M = 64$ and $M = 128$ are fairly similar. Also setting proper values for $\lambda_{max}, \sigma_{max}$ is key to the improved results. The left-most panels show that with fixed σ_{max} values (or Ensemble-U), the preferred conditional variance cap values λ_{max} are fairly small (but still larger than 0 which corresponds to a point estimate for $\boldsymbol{W}$ given $\boldsymbol{U}$). For σ_{max} which controls variance in $\boldsymbol{U}$ space, we see from the top middle panel

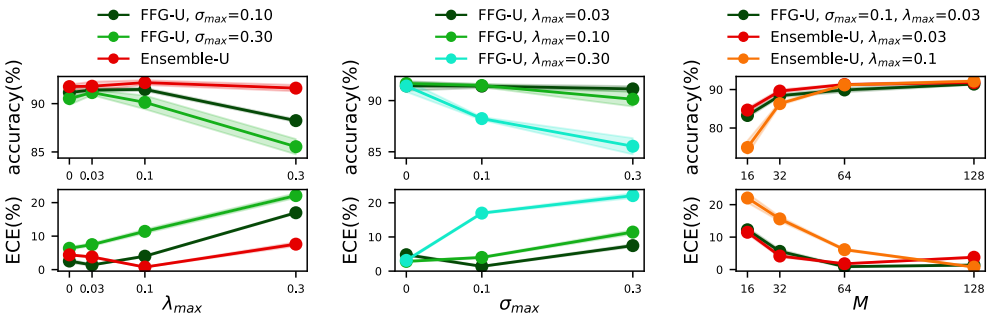


Figure 4.10: Ablation study: average CIFAR-10 accuracy (↑) and ECE (↓) for the inducing weight methods on ResNet-18. In the first two columns $M = 128$ for $\boldsymbol{U}$ dimensions. For $\lambda_{max}, \sigma_{max} = 0$ we use point estimates for $\boldsymbol{U}, \boldsymbol{W}$ respectively.

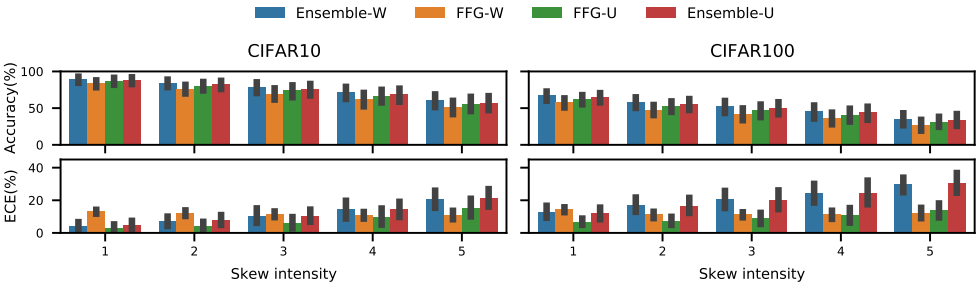


Figure 4.11: Mean±two errs. for Acc↑ and ECE↓ on corrupted CIFAR [120].

that the accuracy metric is fairly robust to σ_{max} as long as λ_{max} is not too large. But for ECE, a careful selection of σ_{max} is required (bottom middle panel).

We run the inducing weight method with the following options:

- Row/column dimensions of $\boldsymbol{U}_l$ (M): $M \in \{16, 32, 64, 128\}$.
  We set $M = M_{in} = M_{out}$ except for the last layer, where we use $M_{in} = M$ and $M_{out} = 10$.

- λ_{max} values for FFG-U and Ensemble-U: $\lambda_{max} \in \{0, 0.03, 0.1, 0.3\}$.
  When $\lambda_{max} = 0$ it means $q(\boldsymbol{W}|\boldsymbol{U})$ is a delta measure centered at the mean of $p(\boldsymbol{W}|\boldsymbol{U})$.

- σ_{max} values for FFG-U: $\sigma_{\text{max}} = \{0, 0.1, 0.3\}$.
  When $\sigma_{max} = 0$ we use a MAP estimate for $\boldsymbol{U}$.

Each experiment is repeated with 5 random seeds to collect the averaged results on a single NVIDIA RTX 2080TI. The models are trained for 100 epochs in total. We first run 50 epochs of maximum likelihood to initialise the model, then run 40 epochs training on the modified variational lower-bound with KL annealing (linear scaling schedule), finally we run 10 epochs of training with the variational lower-bound (i.e. no KL annealing). We use the Adam optimiser with learning rate $3e-4$ and the default β_1, β_2 parameters in PyTorch's implementation.

4.2.4.4 Robustness, out-of-distribution detection and pruning

To investigate the models' robustness to distribution shift, we compute predictions on corrupted CIFAR datasets [120] after training on clean data. Fig. 4.11 shows

Table 4.7: OOD detection metrics for Resnet-50 trained on CIFAR10/100.

In-dist → OOD	C10 → C100		C10 → SVHN		C100 → C10		C100 → SVHN	
Method / Metric	AUROC	AUPR	AUROC	AUPR	AUROC	AUPR	AUROC	AUPR
Deterministic	.84±.00	.80±.00	.93±.01	.85±.01	.74±.00	.74±.00	.81±.01	.72±.02
Ensemble-W	.89	.89	.95	.92	.78	.79	.86	.78
FFG-W	.88±.00	.90±.00	.90±.01	.86±.01	.76±.00	.79±.00	.80±.01	.69±.01
FFG-U	.89±.00	.91±.00	.94±.01	.91±.01	.77±.00	.79±.00	.83±.01	.74±.01
Ensemble-U	.90±.00	.91±.00	.93±.00	.91±.00	.77±.00	.79±.00	.82±.01	.72±.02

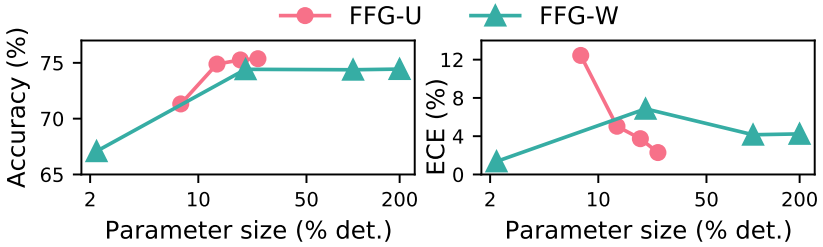


Figure 4.12: CIFAR100 pruning accuracy(↑) and ECE(↓) as a function of the number of parameters relative to a deterministic network. Rightmost points are without pruning.

accuracy and ECE results for the ResNet-50 models. Ensemble-W is the most accurate model across skew intensities, while FFG-W, though performing well on clean data, returns the worst accuracy under perturbation. The inducing weight methods perform competitively to Ensemble-W with Ensemble-U being slightly more accurate than FFG-U as on the clean data. For ECE, FFG-U outperforms Ensemble-U and Ensemble-W, which are similarly calibrated. Interestingly, while the accuracy of FFG-W decays quickly as the data is perturbed more strongly, its ECE remains roughly constant.

Tab. 4.7 further presents the utility of the maximum predicted probability for out-of-distribution (OOD) detection. The metrics are the area under the receiver operator characteristic (AUROC) and the precision-recall curve (AUPR). The inducing-weight methods perform similarly to Ensemble-W; all three outperform FFG-W and deterministic networks across the board.

4.2.4.5 Parameter pruning

We further investigate pruning as a pragmatic alternative for more parameter-efficient inference. For FFG-U, we prune entries of the Z matrices, which contribute the largest number of parameters to the inducing methods, with the smallest magnitude.

Table 4.8: Pruning in-distribution uncertainty results for Resnet-50. The percentage refers to the weights left for FFG-W and the number of Z parameters for FFG-U.

Method	CIFAR10				CIFAR100			
	Acc. ↑	NLL ↓	ECE ↓	Brier ↓	Acc. ↑	NLL ↓	ECE ↓	Brier ↓
FFG-W (100%)	94.13±0.08	0.18±0.00	0.50±0.06	0.09±0.00	74.44±0.27	1.01±0.01	4.24±0.10	0.35±0.00
FFG-W (50%)	94.07±0.03	0.18±0.00	0.40±0.04	0.09±0.00	74.38±0.21	1.02±0.01	4.15±0.09	0.36±0.00
FFG-W (10%)	94.17±0.06	0.18±0.00	0.58±0.02	0.09±0.00	74.42±0.23	1.10±0.01	6.86±0.08	0.36±0.00
FFG-W (1%)	93.60±0.09	0.19±0.00	0.80±0.06	0.09±0.00	67.08±0.33	1.19±0.01	1.36±0.18	0.44±0.00
FFG-W (0.1%)	58.59±0.75	1.15±0.02	7.33±0.23	0.55±0.01	10.66±0.43	3.97±0.03	2.80±0.15	0.96±0.00
FFG-U (100%)	94.40±0.05	0.17±0.00	0.64±0.06	0.08±0.00	75.37±0.09	0.92±0.01	2.29±0.39	0.34±0.00
FFG-U (75%)	94.45±0.05	0.18±0.00	2.19±0.11	0.09±0.00	75.26±0.07	0.93±0.00	3.74±0.67	0.35±0.00
FFG-U (50%)	94.31±0.07	0.18±0.00	2.31±0.09	0.09±0.00	74.89±0.07	0.94±0.00	5.04±0.77	0.35±0.00
FFG-U (25%)	93.34±0.03	0.22±0.00	4.83±0.09	0.11±0.00	71.32±0.16	1.09±0.01	12.44±0.90	0.42±0.00

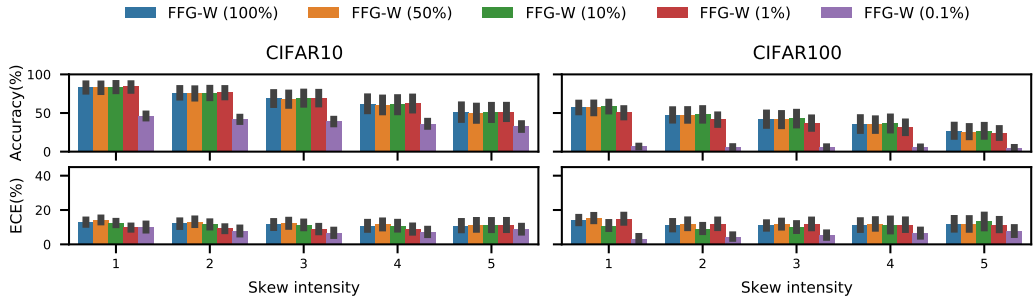


Figure 4.13: Accuracy (↑) and ECE (↓) on corrupted CIFAR for pruning FFG-W. We show the mean and two standard errors for each metric on the 19 perturbations provided in [120].

For FFG-W we follow Graves [100] in setting different fractions of $\boldsymbol{W}$ to 0 depending on their variational mean-to-variance ratio and repeat the previous experiments after fine-tuning the distributions on the remaining variables. We stress that, unlike FFG-U, the FFG-W pruning corresponds to a post-hoc change of the probabilistic model and no longer performs inference in the original weight-space.

For FFG-W, pruning 90% of the parameters (leaving 20% of parameters as compared to its deterministic counterpart) worsens the ECE, in particular on CIFAR100, see Fig. 4.12 for a plot of accuracy and ECE as a function of the number of parameters. Further pruning to 1% worsens the accuracy and the OOD detection results as well. On the other hand, pruning 50% of the Z matrices for FFG-U reduces the parameter count to 13.2% of a deterministic net, at the cost of only slightly worse calibration. The results of pruning different fractions of the weights can be found in Tab. 4.8 for the in-distribution uncertainty evaluation for Resnet-50 and the OOD detection in Tab. 4.10.

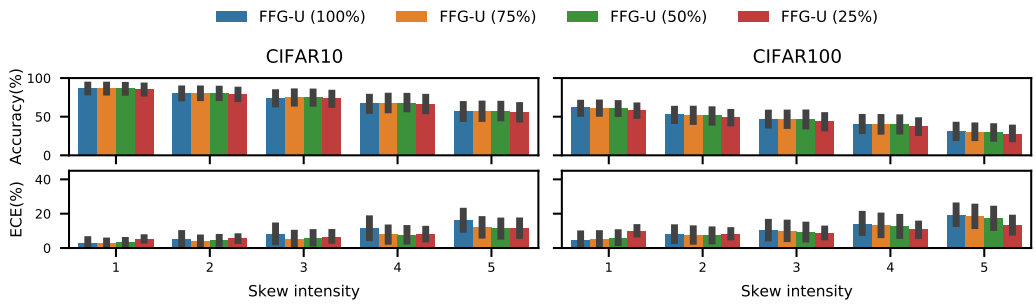


Figure 4.14: Accuracy (↑) and ECE (↓) on corrupted CIFAR for pruning FFG-U. We show the mean and two standard errors for each metric on the 19 perturbations provided in [120].

For FFG-W we find that pruning up to 90% of the weights only worsens ECE and NLL on the more difficult CIFAR100 datasets. Pruning 99% of the weights worsens accuracy and OOD detection, but interestingly improves ECE on CIFAR100, where accuracy is noticeably worse.

Pruning 25% and 50% of the Z parameters in FFG-U results in a total parameter count of $4,408,790$ and $3,106,678$, i.e. 18.7 and 13.2% of the deterministic parameters respectively on ResNet-50, see also Tab. 4.9. Up to pruning 50% of the Z parameters, we find that only ECE becomes slightly worse, although on CIFAR100 it is still better than the ECE for FFG-W at 100% of the weights. Other metrics are not affected neither on the in-distribution uncertainty or OOD detection, except for a minor drop in accuracy.

Table 4.9: Pruning parameter counts for keeping fractions of the weights in FFG-W and the Z parameters in FFG-U.

Method	Abs. param. count	rel. size (%)
FFG-W (100%)	$46,988,564$	199.8
FFG-W (50%)	$23,520,852$	100
FFG-W (10%)	$4,746,682$	20.2
FFG-W (1%)	$522,494$	2.2
FFG-W (0.1%)	$100,075$	0.4
FFG-U (100%)	$5,710,902$	24.28
FFG-U (75%)	$4,408,790$	18.7
FFG-U (50%)	$3,106,678$	13.2
FFG-U (25%)	$1,804,566$	7.7

Table 4.10: Pruning OOD detection metrics for Resnet-50 trained on CIFAR10/100.

In-dist → OOD	C10 → C100		C10 → SVHN		C100 → C10		C100 → SVHN	
Method / Metric	AUROC	AUPR	AUROC	AUPR	AUROC	AUPR	AUROC	AUPR
FFG-W (100%)	.88±.00	.90±.00	.90±.01	.86±.01	.76±.00	.79±.00	.80±.01	.69±.01
FFG-W (50%)	.88±.00	.90±.00	.90±.00	.86±.00	.76±.00	.79±.00	.79±.01	.69±.01
FFG-W (10%)	.88±.00	.90±.00	.90±.01	.87±.01	.76±.00	.79±.00	.79±.01	.69±.01
FFG-W (1%)	.88±.00	.89±.00	.90±.01	.86±.01	.72±.00	.75±.00	.71±.02	.57±.02
FFG-W (0.1%)	.65±.01	.65±.01	.38±.03	.24±.03	.56±.01	.59±.01	.31±.04	.21±.02
FFG-U (100%)	.89±.00	.91±.00	.94±.01	.91±.01	.77±.00	.79±.00	.83±.01	.74±.01
FFG-U (75%)	.89±.00	.91±.00	.94±.00	.91±.00	.77±.00	.79±.00	.82±.01	.72±.02
FFG-U (50%)	.89±.00	.91±.00	.93±.00	.91±.00	.77±.00	.79±.00	.82±.01	.72±.02
FFG-U (25%)	.88±.00	.90±.00	.92±.01	.88±.01	.75±.00	.77±.00	.82±.02	.72±.03

4.2.4.6 Implementation details

We base our implementation on the Resnet class in torchvision [269], replacing the input convolutional layer with a 3×3 kernel size and removing the max-pooling layer. We train the deterministic network on CIFAR-10 using Adam with a learning rate of 3×10^{-4} for 200 epochs. On CIFAR-100 we found SGD with a momentum of 0.9 and initial learning rate of 0.1 decayed by a factor of 0.1 after 60, 120 and 160 epochs to lead to better accuracies. The ensemble is formed of the five deterministic networks trained with different random seeds.

For FFG-W we initialised the mean parameters using the default initialisation in listing for the corresponding deterministic layers. The initial standard deviations are set to 10^{-4}. We train using Adam for 200 epochs on CIFAR-10 with a learning rate of 3×10^{-4}, and 300 epochs on CIFAR-100 with an initial learning rate of 10^{-3}, decaying by a factor of 0.1 after 200 epochs. On both datasets we only use the negative log likelihood part of the variational lower bound for the first 100 epochs as initialisation to the maximum likelihood parameter and then anneal the weight of the kl term linearly over the following 50 epochs. For the prior we use a standard Gaussian on all weights and biases and restrict the standard deviation of the posterior to be at most $\sigma_{max} = 0.1$. We also experimented with a larger upper limit of $\sigma_{max} = 0.3$, but found this to negatively affect both accuracy and calibration.

All the $\boldsymbol{U}$-space approaches use Gaussian priors $p(\text{vec}(\boldsymbol{W}_l)) = \mathcal{N}(\boldsymbol{0}, 1/\sqrt{d_{in}}\boldsymbol{I})$, motivated by the connection to GPs. Hyperparameter and optimisation details for

the inducing weight methods on CIFAR-10 are discussed below in the details on the ablation study. We train all methods using Adam for 300 epochs with a learning rate of 10^{-3} for the first 200 epochs and then decay by a factor of 10. For the initial 100 epochs we train without the KL-term of the ELBO and then anneal its weight linearly over the following 50 epochs. For the tables and figures in the main text, we set $\lambda_{max} = 0.1$ for Ensemble-U on both datasets, and $\sigma_{max} = 0.1, \lambda_{max} = 0.03$ on CIFAR-10 for FFG-U. We initialise the entries of the Z matrices by sampling from a zero-mean Gaussian with variance $\frac{1}{M}$ and set the diagonal entries of the D matrices to 10^{-3}. For FFG-U we initialise the mean of the variational Gaussian posterior in $\boldsymbol{U}$-space by sampling from a standard Gaussian and set the initial variances to 10^{-3}. For Ensemble-U initialisation, we draw an $M \times M$ shaped sample from a standard Gaussian that is shared across ensemble members and add independent Gaussian noise with a standard deviation of 0.1 for each member. We use an ensemble size of 5. During optimisation, we draw 1 MC samples per update step for both FFG-U and Ensemble-U (such that each ensemble member is used once). For testing we use 20 MC samples for all variational methods. We fit BatchNorm parameters by minimising the negative log likelihood.

For the pruning experiments, we take the parameters from the corresponding full runs, set a fixed percentage of the weights to be deterministically 0 and fine-tune the remaining weights with a new optimizer for 50 epochs. We use Adam with a learning rate of 10^{-4}. For FFG-W we select the weights with the smallest ratio of absolute mean to standard deviation in the approximate posterior, and for FFG-U the Z parameters with the smallest absolute value.

4.2.5 Related Work

4.2.5.1 Parameter-efficient uncertainty quantification methods

Recent research has proposed Gaussian posterior approximations for BNNs with efficient covariance structure [289, 373, 242]. The inducing weight approach differs from these in introducing structure via a hierarchical posterior with low-dimensional auxiliary variables. Another line of work reduces the memory overhead via efficient parameter sharing [215, 353, 324, 62]. The third category of work considers a

hybrid approach, where only a selective part of the neural network receives Bayesian treatments, and the other weights remain deterministic [33, 53]. However, both types of approaches maintain a "mean parameter" for the weights, making the memory footprint at least that of storing a deterministic neural network. Instead, our approach shares parameters via the augmented prior with efficient low-rank structure, *reducing* the memory use compared to a deterministic network. In a similar spirit, Izmailov et al. [151] perform inference in a d-dimensional sub-space obtained from PCA on weights collected from an SGD trajectory. But this approach does not leverage the layer-structure of neural networks and requires $d\times$ memory of a single network.

As a side note, mean-field VI approaches have shown success in network pruning but only in terms of maintaining a minimum accuracy level [100, 216, 111]. To the best of our knowledge, our empirical study presents the first evaluation for pruning methods in maintaining uncertainty estimation quality.

4.2.5.2 Sparse GP and function-space inference

As BNNs and GPs are closely related [250, 230, 196], recent efforts have introduced GP-inspired techniques to BNNs [218, 321, 165, 258]. Compared to weight-space inference, function-space inference is appealing as its uncertainty is more directly relevant for predictive uncertainty estimation. While the inducing weight approach performs computations in weight-space, Section 4.2.6 establishes the connection to function-space posteriors. Our approach is related to sparse deep GP methods with $\boldsymbol{U}_c$ having similar interpretations as inducing outputs in e.g. Salimbeni and Deisenroth [300]. The major difference is that $\boldsymbol{U}$ lies in a low-dimensional space, projected from the pre-activation output space of a network layer.

4.2.5.3 Priors on neural network weights

Hierarchical priors for weights has also been explored [216, 181, 13, 91, 159]. However, we emphasise that $\tilde{p}(\boldsymbol{W},\boldsymbol{U})$ is a *pseudo prior*, i.e. a prior that is constructed to *assist posterior inference* (rather than to *improve model design* by more accurately reflecting our beliefs about the distribution of the weights). Indeed, parameters associated with the inducing weights are optimisable for improving posterior approximations. Our approach can be adapted to other priors, e.g. for a Horseshoe prior

$p(\boldsymbol{\theta},\boldsymbol{\nu}) = p(\boldsymbol{\theta}|\boldsymbol{\nu})p(\boldsymbol{\nu}) = \mathcal{N}(\boldsymbol{\theta};0,\boldsymbol{\nu}^2)C^+(\boldsymbol{\nu};0,1)$, the pseudo prior can be defined as $\tilde{p}(\boldsymbol{\theta},\boldsymbol{\nu},\mathbf{a}) = \tilde{p}(\boldsymbol{\theta}|\boldsymbol{\nu},\mathbf{a})\tilde{p}(\mathbf{a})p(\boldsymbol{\nu})$ such that $\int \tilde{p}(\boldsymbol{\theta}|\boldsymbol{\nu},\mathbf{a})\tilde{p}(\mathbf{a})d\mathbf{a} = p(\boldsymbol{\theta}|\boldsymbol{\nu})$. In general, pseudo priors have found broader success in Bayesian computation [41].

4.2.6 A function-space perspective on inducing weights

We present the proposed approach again but from a function-space inference perspective. Assume a neural network layer with weight $\boldsymbol{W}$ computes the following transformation of the input $\mathbf{X} = [\boldsymbol{x}_1,...,\boldsymbol{x}_N], \boldsymbol{x}_i \in \mathbb{R}^{d_{in}\times 1}$

$$\mathbf{F} = \boldsymbol{W}\mathbf{X},\ \mathbf{H} = g(\mathbf{F}), \quad \boldsymbol{W} \in \mathbb{R}^{d_{out}\times d_{in}}, \mathbf{X} \in \mathbb{R}^{d_{in}\times N},$$

where $g(\cdot)$ is the non-linearity. Therefore the Gaussian prior $p(W) = \mathcal{N}(\mathbf{0},\sigma^2\boldsymbol{I})$ induces a Gaussian distribution on the linear transformation output $\mathbf{F}$, in fact each of the rows in $\mathbf{F} = [\mathbf{f}_1,...,\mathbf{f}_{d_{out}}]^\top, \mathbf{f}_i \in \mathbb{R}^{N\times 1}$ has a Gaussian process form with linear kernel

$$\mathbf{f}_i|\mathbf{X} \sim \mathcal{GP}(\mathbf{0},\mathbf{K}_{\mathbf{XX}}), \quad \mathbf{K}_{\mathbf{XX}}(m,n) = \sigma^2\boldsymbol{x}_m^\top\boldsymbol{x}_n. \tag{4.35}$$

Performing inference on $\mathbf{F}$ directly has $\mathcal{O}(N^3 + d_{out}N^2)$ cost, so a sparse approximation is needed. Slightly different from the usual approach, we introduce "scaled noisy inducing outputs" $\boldsymbol{U}_c = [\mathbf{u}_1^c,...,\mathbf{u}_{d_{out}}^c]^\top \in \mathbb{R}^{d_{out}\times M_{in}}$ in the following way, using shared inducing inputs $\boldsymbol{Z}_c^\top \in \mathbb{R}^{d_{in}\times M_{in}}$

$$p(\mathbf{f}_i,\hat{\mathbf{u}}_i|\mathbf{X}) = \mathcal{GP}\left(\mathbf{0},\begin{pmatrix}\mathbf{K}_{\mathbf{XX}} & \mathbf{K}_{\mathbf{XZ}_c}\\ \mathbf{K}_{\mathbf{Z}_c\mathbf{X}} & \mathbf{K}_{\mathbf{Z}_c\mathbf{Z}_c}\end{pmatrix}\right),$$
$$p(\mathbf{u}_i^c|\hat{\mathbf{u}}_i) = \mathcal{N}\left(\frac{\hat{\mathbf{u}}_i}{\sigma_c},\sigma_r^2\boldsymbol{D}_c^2\right),$$

with $\mathbf{K}_{\mathbf{Z}_c\mathbf{X}} = \sigma^2\boldsymbol{Z}_c\mathbf{X}$ and $\mathbf{K}_{\mathbf{Z}_c\mathbf{Z}_c} = \sigma^2\boldsymbol{Z}_c\boldsymbol{Z}_c^\top$. By marginalising out the "noiseless inducing outputs" $\hat{\mathbf{u}}_i$, we have the joint distribution $p(\mathbf{f}_i,\mathbf{u}_i)$ as

$$p(\mathbf{u}_i^c) = \mathcal{N}(\mathbf{0},\sigma_r^2\boldsymbol{\Psi}_c),\ \boldsymbol{\Psi}_c = \boldsymbol{Z}_c\boldsymbol{Z}_c^\top + \boldsymbol{D}_c^2,$$
$$p(\mathbf{f}_i|\mathbf{X},\mathbf{u}_i^c) = \mathcal{N}(\sigma_c\sigma^{-2}\mathbf{K}_{\mathbf{XZ}_c}\boldsymbol{\Psi}_c^{-1}\mathbf{u}_i^c, \mathbf{K}_{\mathbf{XX}} - \sigma^{-2}\mathbf{K}_{\mathbf{XZ}_c}\boldsymbol{\Psi}_c^{-1}\mathbf{K}_{\mathbf{Z}_c\mathbf{X}}).$$

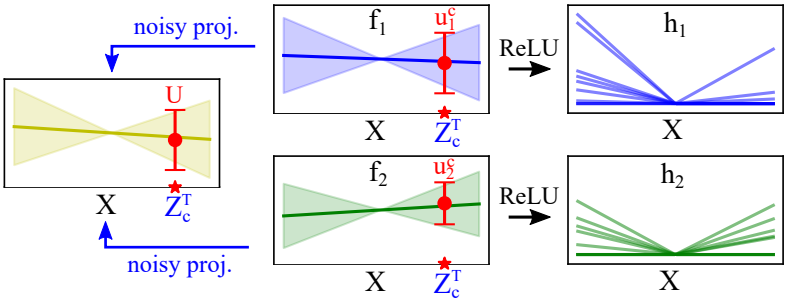


Figure 4.15: Visualising $\boldsymbol{U}$ variables in pre-activation spaces.

Collecting all the random variables in matrix forms, this leads to

$$\begin{aligned} p(\boldsymbol{U}_c) &= \mathcal{MN}(\mathbf{0}, \sigma_r^2 \boldsymbol{I}, \boldsymbol{\Psi}_c), \\ p(\mathbf{F}|\mathbf{X}, \boldsymbol{U}_c) &= \mathcal{MN}(\sigma_c \sigma^{-2} \boldsymbol{U}_c \boldsymbol{\Psi}_c^{-1} \mathbf{K}_{\mathbf{Z}_c\mathbf{X}}, \sigma_r^2 \boldsymbol{I}, \sigma_r^{-2}(\mathbf{K}_{\mathbf{XX}} - \sigma^{-2} \mathbf{K}_{\mathbf{XZ}_c} \boldsymbol{\Psi}_c^{-1} \mathbf{K}_{\mathbf{Z}_c\mathbf{X}})) \\ &= \mathcal{MN}(\boldsymbol{U}_c \boldsymbol{\Psi}_c^{-1} \sigma_c \boldsymbol{Z}_c \mathbf{X}, \sigma_r^2 \boldsymbol{I}, \mathbf{X}^\top \sigma_c^2 (\boldsymbol{I} - \boldsymbol{Z}_c^\top \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c) \mathbf{X}). \end{aligned} \tag{4.36}$$

Also recall from conditioning rules of matrix normal distributions, we have that

$$p(\boldsymbol{W}|\boldsymbol{U}_c) = \mathcal{MN}\left(\boldsymbol{U}_c \boldsymbol{\Psi}_c^{-1} \sigma_c \boldsymbol{Z}_c, \sigma_r^2 \boldsymbol{I}, \sigma_c^2 (\boldsymbol{I} - \boldsymbol{Z}_c^\top \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c)\right).$$

Since for $\boldsymbol{W} \sim \mathcal{MN}(\mathbf{M}, \boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2)$ we have $\boldsymbol{W}\mathbf{X} \overset{d}{\sim} \mathcal{MN}(\mathbf{M}\mathbf{X}, \boldsymbol{\Sigma}_1, \mathbf{X}^\top \boldsymbol{\Sigma}_2 \mathbf{X})$, this immediately shows that $p(\mathbf{F}|\mathbf{X}, \boldsymbol{U}_c)$ is the push-forward distribution of $p(\boldsymbol{W}|\boldsymbol{U}_c)$ for the operation $\mathbf{F} = \boldsymbol{W}\mathbf{X}$. In other words

$$\mathbf{F} \sim p(\mathbf{F}|\mathbf{X}, \boldsymbol{U}_c) \quad \Leftrightarrow \quad \boldsymbol{W} \sim p(\boldsymbol{W}|\boldsymbol{U}_c),\ \mathbf{F} = \boldsymbol{W}\mathbf{X}.$$

As $\{\mathbf{u}_i^c\} = \boldsymbol{U}_c$ are the "noisy" versions of $\{\hat{\mathbf{u}}_i\}$ in $\mathbf{f}$ space, they can be viewed as "scaled noisy inducing outputs" in $\mathbf{F}$ space, see the red bars in the 2nd row of Fig. 4.15 with $\sigma_c = 1$. As the inducing weights $\boldsymbol{U}$ are the focus of our analysis here, we conclude that this specific choice of σ_c is without loss of generality.

So far the $\boldsymbol{U}_c$ variables assist the posterior inference to capture correlations across functions values of different inputs. Up to now the function values remain independent across output dimensions, which is also reflected by the diagonal row

covariance matrices in the above matrix normal distributions. As in neural networks the output dimension can be fairly large (e.g. $d_{out} = 1000$), to further improve memory efficiency, we proceed to project the *column vectors* of $\boldsymbol{U}_c$ to an M_{out} dimensional space with $M_{out} \ll d_{out}$. This dimension reduction step is done with a generative approach, similar to probabilistic PCA [330]

$$\begin{aligned} \boldsymbol{U} &\sim \mathcal{MN}(0, \boldsymbol{\Psi}_r, \boldsymbol{\Psi}_c), \\ \boldsymbol{U}_c|\boldsymbol{U} &\sim \mathcal{MN}(\sigma_r \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \boldsymbol{U}, \sigma_r^2 (\boldsymbol{I} - \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \boldsymbol{Z}_r), \boldsymbol{\Psi}_c). \end{aligned} \tag{4.37}$$

Note that the column covariance matrices in the above two matrix normal distributions are the same, and the conditional sampling procedure is done by a linear transformation of the columns in $\boldsymbol{U}$ plus noise terms. Again from the marginalisation and conditioning rules of matrix normal distributions, we have that the full joint distribution Eq. 4.21, after proper marginalisation and conditioning, returns

$$\begin{aligned} p(\boldsymbol{U}) &= \mathcal{MN}(0, \boldsymbol{\Psi}_r, \boldsymbol{\Psi}_c), \\ p(\boldsymbol{U}_c|\boldsymbol{U}) &= \mathcal{MN}(\sigma_r \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \boldsymbol{U}, \sigma_r^2 (\boldsymbol{I} - \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \boldsymbol{Z}_r), \boldsymbol{\Psi}_c). \end{aligned}$$

This means $\boldsymbol{U}$ can be viewed as "projected noisy inducing points" for the GP $p(\mathbf{F})$, whose corresponding "inducing inputs" are row vectors in $\boldsymbol{Z}_c$, see the red bars in Fig. 4.15. Similarly, column vectors in $\boldsymbol{U}_r\mathbf{X}$ can be viewed as the noisy projections of the column vectors in $\mathbf{F}$, in other words $\boldsymbol{U}_r$ can also be viewed as "neural network weights" connecting the data inputs $\mathbf{X}$ to the projected output space that $\boldsymbol{U}$ lives in.

As for the variational objective, since $q(\boldsymbol{W}|\boldsymbol{U})$ and $p(\boldsymbol{W}|\boldsymbol{U})$ only differ in the scale of the covariance matrices, it is straightforward to show that the push-forward distribution $q(\boldsymbol{W}|\boldsymbol{U}) \to q(\mathbf{F}|\mathbf{X}, \boldsymbol{U})$ has the same mean as $p(\mathbf{F}|\mathbf{X}, \boldsymbol{U})$, but with a different covariance matrix that scales $p(\mathbf{F}|\mathbf{X}, \boldsymbol{U})$'s covariance matrix by λ^2. As the operation $\mathbf{F} = \boldsymbol{W}\mathbf{X}$ maps $\boldsymbol{W} \in \mathbb{R}^{d_{out} \times d_{in}}$ to $\mathbf{F} \in \mathbb{R}^{d_{out} \times N}$, this means the conditional

KL is scaled up/down, depending on whether $N \geq d_{in}$ or not

$$\mathbb{KL}[q(\mathbf{F}|\mathbf{X},\boldsymbol{U})\,||\,p(\mathbf{F}|\mathbf{X},\boldsymbol{U})] = \frac{N}{d_{in}} R(\lambda),$$
$$R(\lambda) := \mathbb{KL}[q(\boldsymbol{W}|\boldsymbol{U})\,||\,p(\boldsymbol{W}|\boldsymbol{U})].$$

In summary, the push-forward distribution of $q(\boldsymbol{W}_{1:L},\boldsymbol{U}_{1:L}) \to q(\mathbf{F}_{1:L},\boldsymbol{U}_{1:L})$ is

$$q(\mathbf{F}_{1:L},\boldsymbol{U}_{1:L}) = \prod_{l=1}^{L} q(\mathbf{F}_l|\mathbf{F}_{l-1},\boldsymbol{U}_l)q(\boldsymbol{U}_l), \quad \mathbf{F}_0 := \mathbf{X},$$

and the corresponding variational lower-bound for $q(\mathbf{F}_{1:L},\boldsymbol{U}_{1:L})$ becomes (with $\mathcal{D} = (\mathbf{X},\mathbf{Y})$)

$$\text{ELBO}(q(\mathbf{F}_{1:L},\boldsymbol{U}_{1:L})) = \mathbb{E}_{q(\mathbf{F}_{1:L})}[\log p(\mathbf{Y}|\mathbf{F}_L)] - \sum_{l=1}^{L}\left(\frac{N}{d_{in}^l} R(\lambda_l) + \mathbb{KL}[q(\boldsymbol{U}_l)\,||\,p(\boldsymbol{U}_l)]\right), \tag{4.38}$$

with d_{in}^l the input dimension of layer l.

Note that

$$\mathbb{E}_{q(\mathbf{F}_{1:L})}[\log p(\mathbf{Y}|\mathbf{F}_L)] = \mathbb{E}_{q(\boldsymbol{W}_{1:L})}[\log p(\mathbf{Y}|\mathbf{X},\boldsymbol{W}_{1:L})] = \mathbb{E}_{q(\boldsymbol{W}_{1:L})}[\log p(\mathcal{D}|\boldsymbol{W}_{1:L})]. \tag{4.39}$$

Comparing equations Eq. 4.30 and Eq. 4.38, we see that the only difference between weight-space and function-space variational objectives comes in the scale of the conditional KL term. Though not investigated in the experiments, we conjecture that it could bring potential advantage to optimise the following variational lower-bound

$$\tilde{\text{ELBO}}(q(\mathbf{F}_{1:L},\boldsymbol{U}_{1:L})) = \mathbb{E}_{q(\mathbf{F}_{1:L})}[\log p(\mathbf{Y}|\mathbf{F}_L)] - \sum_{l=1}^{L}\left(\beta_l R(\lambda_l) + \mathbb{KL}[q(\boldsymbol{U}_l)\,||\,p(\boldsymbol{U}_l)]\right), \tag{4.40}$$

$$\beta_l = \min(1, \frac{N}{d_{in}^l}).$$

The intuition is that, as uncertainty is expected to be lower when $N \geq d_{in}$, it makes sense to use $\beta = 1 \leq N/d_{in}$ to reduce the regularisation effect introduced by the KL term. In other words, this allows the variational posterior to focus more on

fitting the data, and in this "large-data" regime over-fitting is less likely to appear. On the other hand, function-space inference approaches (such as GPs) often return better uncertainty estimates when trained on small data ($N < d_{in}$). So choosing $\beta = N/d_{in} < 1$ in this case would switch to function-space inference and thereby improving uncertainty quality potentially. In the CIFAR experiments, the usage of convolutional filters leads to the fact that $N \geq d_{in}^l$ for all ResNet layers. Therefore in those experiments $\beta_l = 1$ for all layers, which effectively falls back to the weight-space objective Eq. 4.30.

4.2.7 Conclusion

We have proposed a parameter-efficient uncertainty quantification framework for neural networks. It augments each of the network layer weights with a small matrix of inducing weights, and by extending Matheron's rule to matrix-normal related distributions, maintains a relatively small run-time overhead compared with ensemble methods. Critically, experiments on prediction and uncertainty estimation tasks show the competence of the inducing weight methods to the state-of-the-art, while reducing the parameter count to under a quarter of a deterministic ResNet-50 before pruning. This represents a significant improvement over prior Bayesian and deep ensemble techniques, which so far have not managed to go below this threshold despite various attempts of matching it closely.

Several directions are to be explored in the future. First, modelling correlations across layers might further improve the inference quality as outlined in Section 4.2.2.3. Second, based on the function-space interpretation of inducing weights, better initialisation techniques can be inspired from the sparse GP and dimension reduction literature. Similarly, this interpretation might suggest other innovative pruning approaches for the inducing weight method, thereby achieving further memory savings. Lastly, the small run-time overhead of our approach can be mitigated by a better design of the inducing weight structure as well as vectorisation techniques amenable to parallelised computation.

Chapter 5

TyXe: Pyro-based Bayesian neural networks for PyTorch

The surge of interest in deep learning over recent years has been fuelled to a large degree by the availability of agile software packages that enable researchers and practitioners alike to quickly experiment with different architectures for their problem setting [269, 1] by providing modular abstractions for automatic differentiation and gradient-based learning. While there has been similarly growing interest in uncertainty estimation for deep neural networks, in particular following the Bayesian paradigm [221, 251], a comparable toolbox of software packages has mostly been missing. A major barrier of entry for the use of BNNs is the large overhead in required code and additional mathematical abstractions compared to stochastic maximum likelihood estimation as commonly performed in deep learning. Moreover, BNNs typically have intractable posteriors, necessitating the use of various approximations when performing inference, which depending on the problem may perform better or worse and frequently require complex bespoke implementations. This oftentimes leads to the development of inflexible small libraries or repetitive code creation that can lack essential "tricks of the trade" for performant BNNs, such as appropriate initialisation schemes, gradient variance reduction [168, 337], or may only provide limited inference strategies to compare outcomes. Even though various general purpose probabilistic programming packages have been built on top of those deep learning libraries (Pyro [22] for PyTorch, Edward2 [337] for Tensorflow), software

```
net = nn.Sequential(nn.Linear(1, 50), nn.Tanh(), nn.Linear(50, 1))
likelihood = tyxe.likelihoods.HomoskedasticGaussian(dataset_size, scale=0.1)
prior = tyxe.priors.IIDPrior(dist.Normal(0, 1))
guide_factory = tyxe.guides.AutoNormal
bnn = tyxe.VariationalBNN(net, prior, likelihood, guide_factory)
```

Listing 5.1: Bayesian non-linear regression setup code example in 5 lines. Line 1 is a standard PyTorch neural network definition, line 2 is the likelihood of the data, corresponding to a data loss object. Line 3 sets the prior and line 4 constructs the approximate posterior distribution on the weights. Line 5 finally brings all components together to set up the BNN. For MCMC, the `guide_factory` would be `HMC` or `NUTS` from `pyro.infer.mcmc` and the `BNN` a `tyxe.MCMC_BNN`.

linking those to BNNs has only been released recently [338] and provides substitutes for Keras' layers [46] to construct BNNs from scratch.

In this chapter we describe TyXe (Greek: chance), a package linking the expressive computational capabilities of PyTorch with the flexible model and inference design of Pyro [22] in service of providing a simple, agile, and useful abstraction for BNNs targeted at PyTorch practitioners. In contrast to Tran et al. [338], our central design principle is to avoid modification of 3rd party code, such as requiring new layer types and re-implementation of architectures utilising them, to enable users to "bayesianise" their existing PyTorch neural networks with minimal overhead.

In the following, we give a high-level overview of the TyXe package and provide examples that illustrate the ease of obtaining uncertainty estimates from pre-defined PyTorch neural networks. We begin with a simple one-dimensional non-linear regression example to give an overview of the features and design. Then we continue by exploring various inference methods for a `torchvision` ResNet and their impact on different uncertainty estimation metrics, highlighting that users can use more complex architectures with existing implementations. In the following example of a graph neural network, we demonstrate that TyXe is compatible not only with native PyTorch packages, but also with 3rd party libraries such as DGL [348]. TyXe can further be used in settings where the loss function does not have a probabilistic interpretation as is often the case e.g. in computer vision. We demonstrate that uncertainty in a pseudo-BNNs can improve generalisation to unseen data in a PyTorch3D [284] rendering example. Finally, we show that our separation

of prior and inference makes it straight-forward to implement variational continual learning [254].

Statement of contributions The work in this chapter was carried out in collaboration with Theofanis Karaletsos. We designed the library and experiments together, and I carried out the coding and implementation with feedback from Theo. We wrote the paper together.

5.1 TyXe by example: non-linear regression

The core components that users interact with in TyXe are the BNN classes. These are intended as wrappers around deterministic PyTorch neural networks inheriting from `nn.Module`. We then leverage Pyro to formulate a probabilistic model over the neural network parameters, in which we perform approximate inference. There are two primary BNN classes with identical interfaces: `tyxe.VariationalBNN` and `tyxe.MCMC_BNN`. Both classes offer a unified workflow of constructing a BNN, fitting it to data and then making predictions.

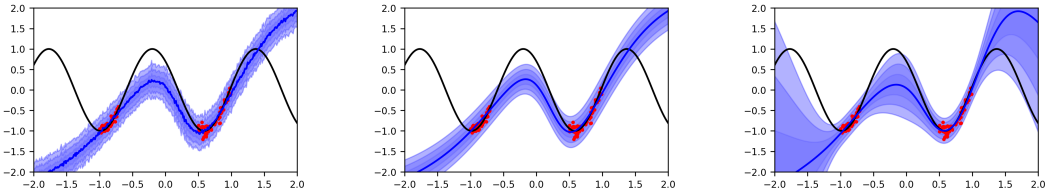
In this section we provide more details on each of these steps along the example of a synthetic one-dimensional non-linear regression dataset. We use the setup from [74] with two clusters of inputs $x_1 \sim \mathcal{U}[-1,-0.7]$, $x_2 \sim \mathcal{U}[0.5,1]$ and $y \sim \mathcal{N}(\cos(4x+0.8), 0.1^2)$.

In all experiments, we use a single Monte Carlo sample for estimating the expected log likelihood during training for variational inference and standard deviations are initialised to 10^{-4} unless stated otherwise. Weight priors are standard normals.

5.1.1 Defining a BNN

A TyXe BNN has four components: a PyTorch neural network, a likelihood for the data, a prior for the weights and a guide factory[1] for the posterior. We describe their signature as well as basic instantiations that we provide below. As seen in Listing 5.1, turning a PyTorch network into a TyXe BNN requires as little as five lines of code.

[1] We follow Pyro's terminology and refer to probabilistic programs drawing approximate posterior samples as "guides".



(a) Local reparameterisation **(b)** Shared weight samples **(c)** HMC

Figure 5.1: Bayesian non-linear regression using the setup from Listing 5.1 and fit using Listing 5.2. Fig. 5.1a wraps the call to `bnn.predict` in the local reparameterisation context with the call to `fit`, Fig. 5.1b does not. Switching between the two is as simple as adapting the indentation of the call to `predict` to be in- or outside the `local_reparameterization` context. Both use the same bnn object with the same approximate posterior. Fig. 5.1c uses `pyro.infer.mcmc.HMC` as guide factory. The shaded area indicates up to three standard deviations from the predictive mean.

5.1.1.1 Network architecture

PyTorch provides a range of classes that facilitate the construction of neural networks, ranging from simple linear or convolutional layers and non-linearities as building blocks to higher-level classes that compose these blocks, e.g. by chaining them together as in the `nn.Sequential` module. A simple regression network on one-dimensional data with one layer of 50 hidden units and a tanh non-linearity, as commonly used for illustration in works on BNNs, can be defined in a single line of code (first line of Listing 5.1). More generally, any neural network in PyTorch is described by the `nn.Module` class, which provides functionalities such as easy composition, parameter and gradient handling, and many more conveniences for neural network researchers and practitioners that have contributed to the wide adoption of this framework. Further, the `torchvision` package implements various modern architectures, such as ResNets [114, 115]. TyXe can also work on top of architectures from 3rd party libraries, such as DGL [348], that derive from `nn.Module`.

Pyro inherits the elegant abstractions for neural networks from PyTorch through its `PyroModule` class, which extends `nn.Module` to allow for instance attributes to be modified by Pyro effect handlers, making it easy to replace `nn.Parameters` with Pyro sample sites. We adopt the `PyroModule` class under the hood to provide a seamless interface between TyXe and PyTorch networks.

5.1.1.2 Prior

At this time, we restrict the probabilistic model definition to specifying priors in weight space. Our prior classes take care of constructing distribution objects that replace the network parameters as `PyroSamples`. One such prior that we provide is an `IIDPrior` which takes a Pyro distribution as argument, such as a `pyro.distributions.Normal(0., 1.)`, applying a standard normal prior over all parameters of the network. We further implement `LayerwiseNormalPrior`, a per-layer Gaussian prior that sets the variance to the inverse of the number of input units as recommended in [250], or analogous to the variance used for weight initialisation in [92, 113] when using the flag `method={"radford", "xavier", "kaiming"}`, respectively. Crucially, we do not require users to set priors for each layer by hand, this is dealt with under the hood automatically within our framework.

Our prior classes accept arguments that allow for certain layers or parameters to be excluded from a Bayesian treatment. Our ResNet example in Section 5.2 passes `hide_module_types=[nn.BatchNorm2d]` to the prior to hide the parameters of the BatchNorm modules. Those parameters stay deterministic and are fit to minimise the log likelihood part of the ELBO.

5.1.1.3 Guide

The guide argument is the only place where the initialisation of our `VariationalBNN` and `MCMC_BNN` differs. `tyxe.VariationalBNN` expects a function that takes a Pyro model as argument and returns a guide, e.g. a `pyro.infer.autoguide`. This is to allow for automatic guide construction for both the network weights and random variables in the likelihood that need to be inferred. For example, if the standard deviation of the data noise in our example was unknown, we could infer it using a LogNormal distribution along side the network weights.

To facilitate local reparameterisation and computation of KL-divergences in closed form, we implement an `AutoNormal` guide, which samples all unobserved sites in the model from a diagonal Normal. This is similar to Pyro's `AutoNormal` autoguide, which constructs an auxiliary joint latent variable with a factorised Gaussian distribution. Variational parameters can be initialised as for autoguides by sampling

```
optim = pyro.optim.Adam({"lr": lr})
with tyxe.poutine.local_reparameterization():
  bnn.fit(loader, n_epochs, optim)
pred_params = bnn.predict(test_data, num_predictions=n)
```

Listing 5.2: Regression fit and predict example with local reparameterisation enabled for training, but not testing.

from the prior/estimating statistics like the prior median, or through additional convenience functions that we provide, such as sampling the means from distributions with variances depending on the numbers of units in the corresponding layers, akin to how deterministic layers are typically initialised. This also permits initialising means to the values of pre-trained networks, which is particularly convenient when converting a deep network into a BNN.

The `tyxe.MCMC_BNN` class expects an MCMC kernel as guide, either HMC [251] or NUTS [135], and runs Pyro's MCMC on the full dataset to obtain (asymptotically unbiased) samples from the posterior.

For both `BNN` classes, arguments to the guide constructor can be passed via `partial` from Python's built-in `functools` module. Listing 5.3 shows an example of this.

5.1.1.4 Likelihood

Our likelihoods are thin wrappers around Pyro's `distributions`, expecting a `dataset_size` argument to correctly scale the KL term when using mini-batches. Specifically we provide Bernoulli, Categorical, HomoskedasticGaussian and HeterosketdasticGaussian likelihoods. Implementing a new likelihood requires a `predictive_distribution(predictions)` method that returns a Pyro distribution object for sampling. Further, it should provide a method for calculating an error estimate for evaluation, such as the squared error for Gaussian models or classification error for discrete models. Hence it is easy to add new likelihoods based on existing distributions, e.g. a Poisson likelihood.

5.1.2 Fitting a BNN

Our BNN class provides a scikit-learn-style `fit` function which runs inference for a given numbers of passes over an `Iterable`, e.g. a PyTorch `DataLoader`. Each element is a length-two tuple, where the first element contains the network inputs (and may be a list) and the second is the likelihood targets, e.g. class labels. The `VariationalBNN` class further requires a Pyro optimiser as input to `fit`.

The `tyxe.VariationalBNN` class runs Stochastic Variational Inference (SVI) [278, 360], which is a popular training algorithm for BNNs, for instance used in [26], based on maximising the evidence lower bound (ELBO). In this case our implementation automatically handles the correct scaling of the KL-term vs. the log likelihood in the ELBO. The `tyxe.MCMC_BNN` provides a compatible interface to Pyro's `MCMC` class.

Listing 5.2 shows a call to `fit`. Besides the data loader and the number of epochs or samples, it is possible to pass in a `callback` function to the `VariationalBNN`, which is invoked after every epoch with the average value of the ELBO over the epoch and can be used e.g. to check the log likelihood of a validation data set. By returning `True`, the callback function can stop training. The `MCMC_BNN` passes any keyword arguments on to Pyro's `MCMC` class.

5.1.3 Predicting with a BNN

The `predict` method returns predictions for a given number of weight samples from the approximate posterior. Listing 5.2 invokes `predict` at the bottom. By default it aggregates the sampled predictions, i.e. averages them. Via `aggregate=False` the sampled predictions can be returned in a stacked tensor. We further implement an `evaluate` method that expects test labels and returns their log likelihood along with an error measure depending on the model, e.g. squared error for Gaussian likelihoods and classification error for Categorical or Binary ones.

5.1.4 Transformations via effect handlers

One crucial component that our library provides is effect handlers[2], for example two popular gradient variance reduction techniques, local reparameterisation[168] and flipout[352]. Local reparameterisation samples the pre-activations of each data point rather than a single weight matrix shared across a mini-batch for factorised Gaussian approximate posteriors over the weights and layers performing linear mappings, such as dense or convolutional layers. Flipout, on the other hand, samples a rank-one matrix of signs per data point, which allows for using distinct weights in a computationally efficient manner in linear operations, if the weights are sampled from a factorised symmetric distribution.

Typically, these are implemented as separate layer classes, e.g. in [338]. This creates an unnecessary redundancy in the code base, since there are now two versions of the same model, which differ only in sampling approaches for gradient estimation at each linear mapping. Moreover, from a probabilistic modeling point of view it is desirable to separate model and inference language explicitly, in order to facilitate reuse of models and inference approaches. Fortunately, Pyro provides an expressive module for effect handling, which we can leverage to modify the computation as required. Specifically, we implement a `LocalReparameterizationMessenger` which marks linear functions called by PyTorch modules, such as `torch.nn.functional.linear`, as effectful in order to modify how linear computations are performed as required. The Messenger maintains references from samples to their distributions and, when a linear function is called in a `local_reparameterization` context on weights from a factorised Gaussian, samples the output from the Gaussian over the result of the linear mapping.

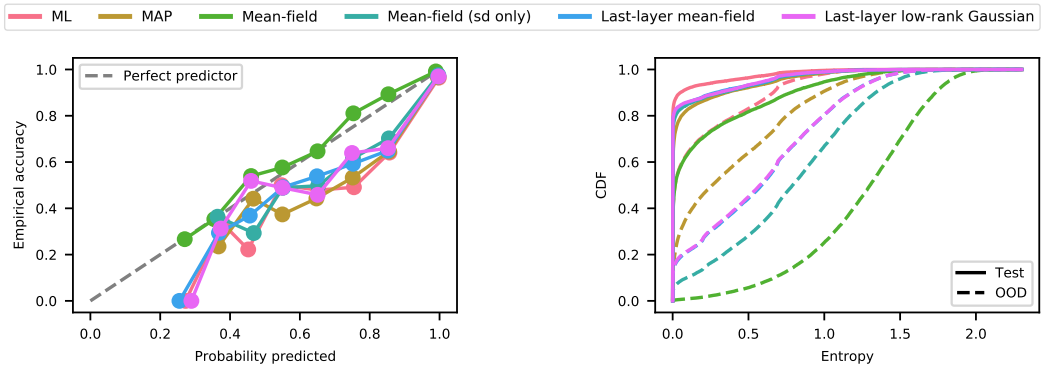
Listing 5.2 calls `fit` in such a context. The call to `predict` could be wrapped too, but the purpose of local reparameterisation and flipout is to reduce gradient variance. As they double the computational cost, we omit them for testing.

5.2 Large-scale vision classification

[2]For an overview of effect handlers, see [272] or specifically Pyro's poutine library [22].

```
1 resnet = torchvision.models.resnet18(pretrained=True)
2 prior = tyxe.priors.IIDPrior(dist.Normal(0, 1), expose_all=False,
3                              hide_module_types=[nn.BatchNorm2d])
4 likelihood = tyxe.likelihoods.Categorical(dataset_size)
5 guide = partial(tyxe.guides.AutoNormal, train_loc=False, init_scale=1e-4,
6   init_loc_fn=tyxe.guides.PretrainedInitializer.from_net(resnet))
7 bayesian_resnet = tyxe.VariationalBNN(resnet, prior, likelihood, guide)
8 ll_prior = tyxe.priors.IIDPrior(dist.Normal(0, 1), expose_all=False,
9   expose_modules=[resnet.fc])  # alternative last-layer prior and guide
10 lr_guide = pyro.infer.autoguide.AutoLowRankMultivariateNormal
```

Listing 5.3: Bayesian ResNet example. Line 1 loads a ResNet with pre-trained parameters from `torchvision`. The prior in lines 2−3 excludes BatchNorm layers, keeping their parameters deterministic. Arguments to the guide are passed with `partial` as in lines 5−6. We show how to set the Gaussian means to the pre-trained weights and only fit the variances, which are initialised to be small. The `BNN` object in line 7 is constructed exactly the same way as in the regression example. Lines 8−10 show an alternative prior that only applies to the final fully-connected layer alongside a Pyro autoguide.



(a) Test calibration.

(b) Test vs. OOD uncertainty.

Figure 5.2: Calibration curves and empirical cumulative density of the entropy of the predictive distribution on test and OOD data for Bayesian Resnet-18 with different inference approaches on CIFAR10 (OOD: SVHN).

The biggest advantage resulting from our choice not to implement bespoke layer classes is that implementations of popular architectures can immediately be turned into their Bayesian counterparts. While implementing the two-layer network from the regression example with Bayesian layers is of course not complicated, writing the code for a modern computer vision architecture, e.g. a ResNet [114, 115], is significantly more cumbersome and error-prone. With TyXe, users can make use of the ResNet implementation available through `torchvision` as shown in Listing 5.3. In this example we further highlight the flexibility of TyXe to only perform

inference over some parameters while keeping others deterministic by excluding `nn.BatchNorm2d` layers from having the prior placed over their parameters.

To showcase how the clean separation of network architecture, prior, guide and likelihood in TyXe facilitates an experimental workflow, we investigate the predictive uncertainty of different inference strategies for a Bayesian ResNet. In Listing 5.3 we define a fully factorised Gaussian guide that fixes the means to the values of pre-trained weights and only fits the variances as parameters. While we would usually want the approximate posterior to be as flexible as possible, it has been observed in the literature [215, 339] that such restrictions can improve the predictive performance of a BNN. We further investigate a mean-field guide where we similarly initialise the means to pre-trained weight values, but do not fix them for optimisation, and restrict the variance of the variational distribution to a maximum of 0.1 to prevent underfitting. Finally we test performing inference in only the final classification layer with a Gaussian guide with either a diagonal or low-rank plus diagonal covariance matrix (also shown in the Listing) while using the pre-trained weights for the previous layers. Switching between these options is easy, with typically only a single or two lines of code differing. As baselines we compare to maximum likelihood (ML) and maximum a-posteriori (MAP). For the full code see `examples/resnet.py` in the supplement.

Fig. 5.2 compares calibration and the entropy of the predictive distributions on test and out-of-distribution (OOD) data. Mean-field (MF) with learned means leads to better calibrated predictions than variants (re-)using point estimates for parts of or the entire network. It best distinguishes test from OOD data as measured by the area under the ROC curve based on the maximum predicted probability and has the lowest ECE and NLL, see Tab. 5.1.

We use the usual data augmentation techniques for CIFAR10 of randomly flipping and cropping the images after padding them with 4 pixels on each dimension and we normalise all channels to have zero-mean and unit-standard deviation. All methods use the Adam optimiser [166]. We train the deterministic inference methods (ML, MAP) for 200 epochs with a learning rate of 10^{-3} and another 100 epochs

Table 5.1: Bayesian ResNet-18 predictive performance.

Inference	NLL↓	Acc.↑(%)	ECE↓(%)	OOD↑
ML	0.33	94.29	4.10	0.78
MAP	0.29	92.14	4.44	0.82
MF (sd only)	0.27	93.66	3.14	0.93
MF	0.20	93.28	0.97	0.94
LL MF	0.35	93.36	3.62	0.89
LL low rank	0.34	93.31	3.75	0.89

with a learning rate of 10^{-4}. All variational methods are trained for 200 epochs with a learning rate of 10^{-3} and we initialise the means to pre-trained ML parameters. We use a rank of 10 for the low-rank plus diagonal posterior and average over 32 samples for predictions on the test and OOD sets. The factorised Gaussian posteriors all use local reparameterisation and we limit the standard deviation of the mean-field posteriors to 0.1.

5.3 Compatibility with external libraries

TyXe is compatible with libraries outside of the native PyTorch ecosystem and classical settings such as classification of i.i.d. images or regression, as long as the networks build on top of `nn.Module`. In this section, we demonstrate this on a semi-supervised node classification example with a graph neural network from the DGL [348] tutorials, as well as a 3D rendering example in PyTorch3D.

5.3.1 Bayesian graph neural networks with DGL

In this section, we extend an example from the DGL tutorials[3] to train a Bayesian graph neural network (GNN) on the Cora dataset. Graph datasets are often semi-supervised, where an entire graph of nodes is provided, but only some of them are labelled. Hence we need a mechanism for preventing unlabelled nodes from contributing to the log likelihood. We combine Pyro's `block` and `mask` poutines to implement the `selective_mask` effect handler, which can wrap the call to `fit` as a context manager as shown in Listing 5.4 and mask out data in the likelihood. The network is taken from the DGL tutorial without change. As it utilises `nn.Linear`, it is compatible with flipout. Prior, guide, likelihood and BNN can be constructed

[3] https://docs.dgl.ai/en/0.5.x/tutorials/models/1_gnn/1_gcn.html

```
1 class GCNLayer(nn.Module):
2   ...
3   def forward(self, graph, x):
4     with graph.local_scope():
5       graph.ndata['h'] = x
6       graph.update_all(gcn_msg, gcn_reduce)
7       h = graph.ndata['h']
8       return self.linear(h)
9
10 class GNN(nn.Module):
11   ...
12   def forward(self, graph, x):
13     x = self.gcn_layer1(graph, x)
14     x = torch.relu(x)
15     return self.gcn_layer2(graph, x)
16 ...
17 bgnn = tyxe.VariationalBNN(gnn, prior, guide, likelihood)
18 loader = [((graph, x), y)]
19 ...
20 with tyxe.poutine.selective_mask(
21     mask=train_mask,
22     expose=["likelihood.data"]):
23   bgnn.fit(loader, optim, n_epochs)
```

Listing 5.4: GNN example. The graph convolutional layer definition in lines 1−9 relies on DGL's graph functionality and is used for the GNN in lines 11−16. The Bayesian GNN can be constructed in lines 18−19 with the exact same prior, guide and likelihood options as in previous examples, while the input data defined in line 20 now consists of a graph and node features. The `selective_mask` in lines 22−24 ensures that only predictions on labelled nodes contribute to the log likelihood when calling `fit` in line 25.

exactly as in the previous examples, see `examples/gnn.py` for the code.

In Tab. 5.2 we report NLLs, accuracies and ECE for ML, MAP and MF. ML leads to overfitting and requires the use of early stopping. Further it suffers from overconfident predictions, which can be mitigated to a degree by the use of variational inference, although not to the same extent as in the image classification example. Bayesian GNNs have only recently been started to be investigated in a few works [375, 109, 217, 186] and we believe that TyXe can be a valuable tool for putting Bayesian inference at the disposal of the graph neural network community.

Following the DGL tutorial, we train ML (and MAP) for 200 iterations with a learning rate of 10^{-2} using Adam. We report the test accuracy at the iteration with lowest validation negative log likelihood. For mean-field, we train for 400 iterations with an initial learning rate of 0.1, which we decay by a factor of 10 every

```
1 nerf_bnn = tyxe.PytorchBNN(nerf_net, prior, guide)
2 optim = torch.optim.Adam(nerf_bnn.pytorch_parameters(dummy_data), lr=1e-3)
3 ...
4 images, rays = renderer(cameras_batch, nerf_bnn)
5 image_loss = calc_loss(images, rays, targets)
6 kl_loss = nerf_bnn.cached_kl_loss
7 loss = image_loss + scale * kl_loss
8 loss.backward()
9 optim.step()
```

Listing 5.5: Bayesian NeRF example. Constructing a `PytorchBNN` is similar to a `VariationalBNN` in lines 1−2 but without the likelihood. No downstream changes except for parameter collection for the PyTorch optimiser in lines 3−5 – which requires a batch of data to trace parameters on a call to the net's forward method – are needed. The `nerf_bnn` can be passed into the PyTorch3D renderer in lines 7−8 as a drop-in replacement for the `nerf_net`. The loss can be calculated as before in lines 9−10, with the possible addition of the KL regulariser in lines 11−12. Automatic differentiation and parameter updates can be performed as in standard PyTorch code in lines 13−14.

100 iterations and we limit the variational standard deviations to 0.3. Means are initialised to the random initialisation of the deterministic network and we draw 8 posterior samples for evaluation. We use 10 bins to calculate the expected calibration error.

5.3.2 Custom losses: Bayesian NeRF with PyTorch3D

Next, we adapt a more complex example on Neural Radiance Fields (NeRF) [237] from the PyTorch3D repository[4] to train a Bayesian NeRF. The loss function does not straight-forwardly correspond to a probabilistic likelihood and is calculated as a custom error function of rendered image and silhouette. Hence there is no suitable likelihood class to implement for TyXe and it is not clear how the the prior

[4] https://github.com/facebookresearch/pytorch3d/blob/master/docs/tutorials/fit_simple_neural_radiance_field.ipynb

Table 5.2: Performance of deterministic and Bayesian GNNs on the Cora dataset. We report the lowest validation NLL along with the test accuracy and ECE at the corresponding epoch (mean and two standard errors over five runs).

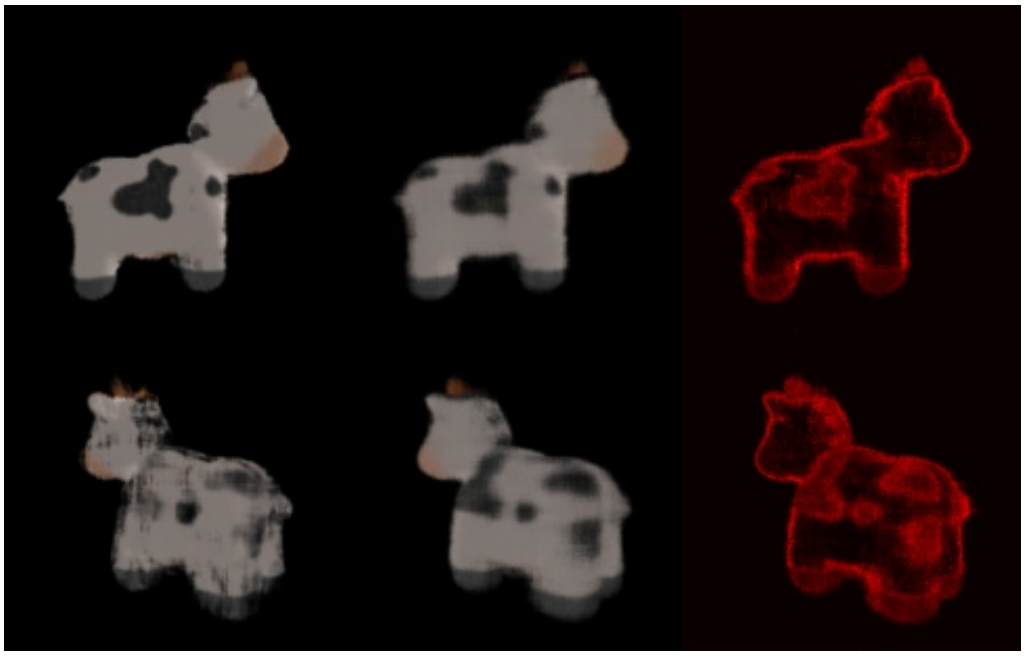
Inference	NLL↓	Acc.↑	ECE↓
ML	$1.01 \pm .04$	75.64 ± 1.28	15.38 ± 0.97
MAP	$0.93 \pm .03$	75.94 ± 0.73	12.78 ± 0.96
MF	$0.77 \pm .02$	78.02 ± 1.00	10.22 ± 1.31

or KL term should be weighed relative to the error. Therefore this example is not Bayesian in the proper sense as a 'posterior' as a product of likelihood and prior does not exist, but demonstrates that the uncertainty of a pseudo-Bayesian variational BNN can still improve the robustness on unseen data.

Specifically, we introduce a more low level `PytorchBNN` class that does not require a likelihood and can be used to directly wrap a PyTorch neural network. It is constructed similarly to `VariationalBNN` with a variational guide factory, but due to the absence of the likelihood does not provide convenience functions such as `fit` or `predict`. Instead, it is intended to serve as a drop-in replacement of the deterministic neural network in a PyTorch-based workflow. The output of the `forward` method corresponds to predictions of the network made with a single Monte Carlo sample from the variational posterior. The corresponding KL penalty term can be accessed through the `cached_kl_loss` attribute and added to the loss. It is updated on every forward pass, i.e. when a sample is drawn from the approximate posterior. The key difference to a regular PyTorch neural network is that since Pyro initialises parameters lazily, we cannot provide a `parameters` method. Instead, optimisable parameters are collected via `pytorch_parameters`, which takes a batch of data to pass through the network for tracing the parameters.

We provide a code snippet in Listing 5.5. We emphasise that parameters are trained with the original PyTorch instead of a Pyro optimiser, further reducing the required changes to the original workflow. The renderer is a PyTorch3D object and uses the Bayesian NeRF object instead of the original PyTorch network. The data-dependent loss is then calculated as before and the KL-divergence of the approximate posterior from the prior on the weights can be added to the objective as a regulariser, possible weighed by some scalar `scale`. The full code can be found in `examples/nerf.py` and is identical to the original notebook for the most part, with only a few lines needing to be modified to adapt it to TyXe, as well as some additional plotting code for visualising the predictive uncertainty.

In the original example, the network is trained to render views of a cow from 360°. We leave out 90° to create a held-out test set. As Fig. 5.3 shows, this



(a) Det. NeRF **(b)** Bayesian NeRF **(c)** Uncertainty

Figure 5.3: PyTorch3D example. Top row was seen during training, bottom row had been excluded. Bayesian NeRF achieves an error of 8.1×10^{-3} on a set of 10 held-out angles, while the error is 9.4×10^{-3} for the deterministic version. Uncertainty visualises the variance across images rendered from different weight samples for the network.

leads to many artifacts and discontinuities with a deterministic net, suggesting overfitting. The pseudo-Bayesian NeRF averages many of these out, and provides helpful measures of uncertainty in form of the variances of the predicted images (right column).

We train the deterministic NeRF with the recommended settings from the tutorial, i.e. $20,000$ iterations with an initial learning rate of 10^{-3}, which is decayed by a factor of 10 for the final 5000 iterations. The Bayesian NeRF uses the same learning rate schedule. Means are initialised to the parameters of the deterministic NeRF and standard deviations to 10^{-2}. We linearly anneal the weight of the KL term over the first $10,000$ iterations to the inverse of the number of RGB values in the colour images plus the number silhouette pixels. We use 8 samples for test predictions and calculate averages and standard deviation over the final image outputs of the renderer.

```
1 bayesian_weights = tyxe.util.pyro_sample_sites(bnn.net)
2 posteriors = bnn.net_guide.get_detached_distributions(bayesian_weights)
3 new_prior = tyxe.priors.DictPrior(posteriors)
4 bnn.update_prior(new_prior)
```

Listing 5.6: Updating the prior of a `BNN` for variational continual learning. Lines 1−2 collect all weights over which we perform inference, lines 3−5 extract the corresponding variational distributions from the guide, lines 6−7 initialise the new prior object mapping parameter names to distribution and line 8 finally updates the BNN's prior.

5.4 Variational continual learning

Finally, we demonstrate how our separation of prior, guide and network architecture allows us to elegantly implement variational continual learning (VCL) [254]. After having set up a BNN as in the previous examples and having trained on the first task, all one needs to do is extract the guide distributions over the weights, construct a new prior object and use it to update the BNNs prior. We show example code for this process in Listing 5.6 and the full implementation can be found in `examples/vcl.py`. Training on the following task can then be conducted as usual with the `fit` method on the current dataset.

In Fig. 5.4 we show the test accuracy across the tasks observed so far after training on each tasks on the classical Split-MNIST and Split-CIFAR benchmarks[371]. We do not make use of coresets as in [254], but this would only require some boilerplate code for creating the coresets prior to training and then fine-tuning on each coreset prior to testing by calling `fit` on them and restoring the state of the Pyro parameter store. As previously reported in the literature, deterministic networks suffer from forgetting on previous tasks, which can be mitigated by using a Bayesian approach such as VCL.

Following the recommendations in [323], we train on each MNIST task for 600 epochs and each CIFAR task for 60. We use Adam with a learning rate of 10^{-3}. The architecture on MNIST is a fully connected network with a hidden layer of 200 units with ReLU non-linearities. The convolutional architecture on CIFAR has two blocks of $Conv - ReLU - Conv - ReLU - Maxpool$ followed by a fully connected layer with 512 units. The convolution layers in the first block have 32, in the second

block 64 channels and all use 3×3 kernels with a stride and padding of 1. The maxpool operation is 2×2 with a stride of 2. We normalise all CIFAR tasks to have zero-mean and unit-standard deviation per channel and do not use any form of data augmentation.

5.5 Related work

The most closely related piece of recent work is Bayesian Layers [338], which extends the layer classes of Keras with the aim of them being usable as drop-in replacements for their deterministic counterpart. This forces the user to modify the code where the network is defined or write their own boilerplate code. Bayesian Layers are currently more general in scope, providing an abstraction over uncertainty over composable functions including normalising flows and Gaussian Process mappings per layer, while at this point we have consciously limited ourselves to weight space uncertainty in neural networks and treat networks holistically rather than per layer.

For PyTorch, PyVarInf[5] provides functionality for turning `nn.Module` instances into BNNs in a similar spirit to TyXe. As it is not backed by a probabilistic programming framework, the choice of prior distributions is limited, inference is restricted to variational factorised Gaussian, sampling tricks such as local reparameterisation are not implemented and MCMC-based inference is not available. More recently, BLiTZ[6] [65] provides variational counterparts to PyTorch's linear, convolutional and some recurrent layers. Networks need to be constructed manually based on those and other types of layers are not supported. Priors are limited to mixtures of up to two Gaussians and inference is performed with a factorised Gaussian without support for gradient variance reduction techniques.

5.6 Conclusion

We have presented TyXe, a Pyro-based library that facilitates a seamless integration of BNNs for uncertainty estimation and continual learning into PyTorch-based

[5] https://github.com/ctallec/pyvarinf
[6] https://github.com/piEsposito/blitz-bayesian-deep-learning

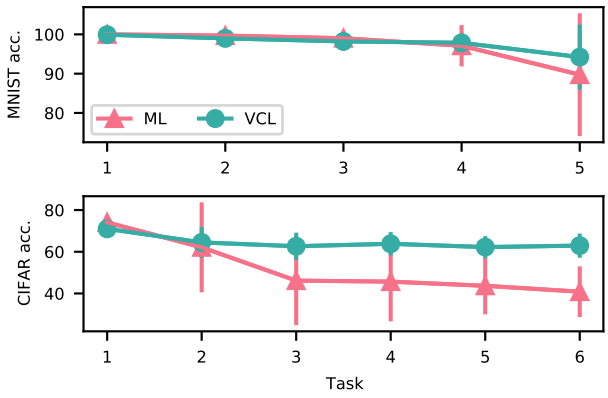


Figure 5.4: Mean accuracy and two standard errors on tasks seen so far for VCL and ML on Split-MNIST and -CIFAR.

workflows. We have demonstrated the flexibility of TyXe with applications based on 3rd-party libraries, ranging from modern deep image classification architectures over graph neural networks to neural radiance fields. TyXe avoids implementing bespoke layer classes and instead leverages and expands on Pyro's powerful effect handler module, resulting in a flexible design that cleanly separates architecture definition, prior, inference, likelihood and sampling logic.

TyXe's choices of variational distributions are currently pragmatic, focused on serving practitioners and researchers interested in generating uncertainty estimates for downstream tasks that will benefit from the improvements offered by standard variational families or HMC over maximum likelihood. Recent work has even argued that mean-field may be sufficient for inference in deep networks [68]. However, we are highly interested in further developing TyXe to support more complex recent approaches and become a tool for Bayesian deep learning research. Its backing by Pyro enables easy extensions for users and developers. We would expect techniques with structured covariance matrices, such as in the previous chapters, [214] as well as hierarchical weight models [215, 160] to be feasible to express within TyXe, although in particular the latter may require some additional abstractions. Nevertheless, we believe that similar to Bayesian Layers [338] TyXe already makes a valuable contribution to the ML software ecosystem, filling the gap of easy-to-use uncertainty estimation for PyTorch.

Chapter 6

Conclusion

6.1 Summary

This thesis presented contributions in three directions of approximate inference in deep neural networks.

Chapter 3 proposed a scalable Laplace approximation for uncertainty estimation and continual learning based on a block-diagonal Kronecker factored approximation of the Hessian. This is an efficient, pragmatic technique that can be applied to existing architectures and training pipelines to add Bayesian uncertainty estimates. Nevertheless, as more recent extensions further demonstrate, this Laplace approximation can form the basis of a competitive approximate inference scheme in modern architectures when combined with optimisation of the hyperparameters w.r.t. the marginal likelihood.

Chapter 4 investigated parameter-efficient inference techniques. The first part explored placing Bernoulli distributions over the weights, for which the samples can cheaply be represented in memory as binary numbers. With a matching Bernoulli prior, variational inference suffers from similar underfitting issues as Gaussian posteriors, however these can be overcome with a MAP-style approach under a hierarchical Beta-Bernoulli prior. The second part developed a method for reducing the parameter count of the variational posteriors by taking inspiration from sparse Gaussian processes. Augmenting the commonly used Gaussian prior in a structured manner and by extending Matheron's rule for efficient sampling to matrix normal

distributions, the approach manages to train Bayesian ResNet-50 models using only 25% of the number of parameters of its deterministic counterpart at matching accuracy being competitive with deep ensembles on the quality of its uncertainty estimates.

Chapter 5 introduced TyXe, a probabilistic programming interface for Bayesian inference in PyTorch neural networks built on top of Pyro. The package allows for flexible experimentation with a range of approximate inference techniques on arbitrary existing architectures by providing cleanly factorised building blocks under a layer-agnostic design.

Below we discuss broader future directions for the field of Bayesian deep learning, see the corresponding chapter and section conclusions for more focused outlooks on the respective techniques.

6.2 Future research

Over the course of the years during which this work has been carried out, uncertainty estimation in deep neural networks has established itself as a core area of machine learning research. Bayesian methods in particular have been a driving force in this direction, offering not only a principled approach for incorporating uncertainty through a probabilistic treatment, but also allowing for overcoming issues such as catastrophic forgetting in a unified way. Much work, including this present one, has focused on developing expressive yet computationally efficient approaches for approximating the posterior as well as possible.

One of the key issues still preventing these Bayesian methods from finding wider adoption is the need for approximating the predictive posterior via Monte Carlo averaging, necessitating multiple passes through a network. With the ever growing size of neural architectures, this creates an often prohibitive overhead, particularly when real-time predictions are needed. One promising recent direction is restricting inference to the output layer, i.e. combining a deterministic feature extractor with a Bayesian last layer. The methods developed in Chapter 3 have been utilised for such pipelines [176, 63] and it will be interesting to see if there

are certain inductive biases or regularisation techniques for the feature extractor to make such approaches particularly effective. Alternatively, propagating the weight uncertainty deterministically through the forward pass could significantly reduce the computational overhead compared to multiple Monte Carlo passes, but may require bespoke techniques for each posterior approximation.

On a more fundamental level, the key question that remains to be answered is whether the predominantly used weight-space inference approaches and Gaussian priors are going to lead to satisfying solutions if inference is accurate enough. Indeed, approximate inference approaches often fall short in particular in terms of discriminative performance when compared to deterministic models and require ad-hoc deviations from a principled attempt at approximating the posterior as well as possible, such as artificially forcing the approximate posterior to be more peaked, to remain competitive. All components of the pipeline, from weight-space priors that fail to express desirable functional priors [159, 76] over the mis-specified likelihoods due to data augmentation [355] to inaccurate inference, have received their share of the blame. However, recent evidence suggests that with typically infeasible MCMC sampling approaches BNNs do outperform deterministic networks and heuristic methods such as ensembles [152] in terms of accuracy. Given the computational difficulty of integrating over a parameters with a dimensionality in the dozens of millions, the overall progress that the field has seen should certainly not be understated and it will be intriguing to see the next developments over the coming years.

Appendix A

Derivation of the pre-activation Hessian recursion

Here, we provide the basic derivation of the factorisation of the diagonal blocks of the Hessian in Eq. Eq. 3.5 and the recursive formula for calculating $\boldsymbol{\mathcal{H}}$ as presented in [29].

The Hessian of a neural network with parameters $\boldsymbol{\theta}$ as defined in the main text has elements

$$H_{i,j} = \frac{\partial^2}{\partial\theta_i \partial\theta_j} E(\boldsymbol{\theta}). \tag{A.1}$$

For a given layer l, the gradient w.r.t. a weight $W^l_{i,j}$ is

$$\frac{\partial E}{\partial W^l_{i,j}} = \sum_k \frac{\partial \mathrm{h}^l_k}{\partial W^l_{i,j}} \frac{\partial E}{\partial \mathrm{h}^l_k} = \mathrm{a}^{l-1}_j \frac{\partial E}{\partial \mathrm{h}^l_i}. \tag{A.2}$$

Keeping l fixed and differentiating again, we find that the per-sample Hessian of that layer is

$$H^l_{(i,j),(m,n)} \equiv \frac{\partial^2 E}{\partial W^l_{i,j} \partial W^l_{m,n}} = \mathrm{a}^{l-1}_j \mathrm{a}^{l-1}_n \mathcal{H}^l_{i,m}, \tag{A.3}$$

where

$$\mathcal{H}^l_{i,m} = \frac{\partial^2 E}{\partial \mathrm{h}^l_i \partial \mathrm{h}^l_m} \tag{A.4}$$

is the pre-activation Hessian.

We can re-express this in matrix notation as a Kronecker product as in Eq.

Eq. 3.5

$$\boldsymbol{H}_l = \frac{\partial^2 E}{\partial \operatorname{vec}(\boldsymbol{W}_l)\partial \operatorname{vec}(\boldsymbol{W}_l)} = \left(\mathbf{a}_{l-1}\mathbf{a}_{l-1}^\top\right) \otimes \boldsymbol{\mathcal{H}}_l. \tag{A.5}$$

The pre-activation Hessian can be calculated recursively as

$$\boldsymbol{\mathcal{H}}_l = \boldsymbol{B}_l \boldsymbol{W}_{l+1}^\top \boldsymbol{\mathcal{H}}_{l+1} \boldsymbol{W}_{l+1} \boldsymbol{B}_l + \boldsymbol{D}_l \tag{A.6}$$

where the diagonal matrices $\boldsymbol{B}$ and $\boldsymbol{D}$ are defined as

$$\boldsymbol{B}_l = \operatorname{diag}(f_l'(\mathbf{h}_l)) \tag{A.7}$$

$$\boldsymbol{D}_l = \operatorname{diag}(f_l''(\mathbf{h}_l)\frac{\partial E}{\partial \mathbf{a}_l}) \tag{A.8}$$

f' and f'' denote the first and second derivative of the transfer function. The recursion is initialised with the Hessian of the error w.r.t. the linear network outputs.

The recursion is derived as follows

$$\begin{aligned}
\mathcal{H}^l_{i,m} &= \frac{\partial}{\partial \mathrm{h}^l_m}\frac{\partial E}{\partial \mathrm{h}^l_i} = \frac{\partial}{\partial \mathrm{h}^l_m}\sum_k \frac{\partial E}{\partial \mathrm{h}^{l+1}_k}\frac{\partial \mathrm{h}^{l+1}_k}{\partial \mathrm{h}^l_i} = \sum_k \frac{\partial}{\partial \mathrm{h}^l_m}\left(\frac{\partial E}{\partial \mathrm{h}^{l+1}_k}\frac{\partial \mathrm{h}^{l+1}_k}{\partial \mathrm{a}^l_i}\frac{\partial \mathrm{a}^l_i}{\partial \mathrm{h}^l_i}\right) \\
&= \sum_k W^{l+1}_{k,i}\frac{\partial}{\partial \mathrm{h}^l_m}\left(\frac{\partial E}{\partial \mathrm{h}^{l+1}_k}\frac{\partial \mathrm{a}^l_i}{\partial \mathrm{h}^l_i}\right) = \sum_k W^{l+1}_{k,i}\left(\frac{\partial \mathrm{a}^l_i}{\partial \mathrm{h}^l_i}\frac{\partial^2 E}{\partial \mathrm{h}^l_m \partial \mathrm{h}^{l+1}_k} + \frac{\partial E}{\partial \mathrm{h}^{l+1}_k}\frac{\partial^2 \mathrm{a}^l_i}{\partial \mathrm{h}^l_i \partial \mathrm{h}^l_m}\right) \\
&= \sum_k W^{l+1}_{k,i}\left(\frac{\partial \mathrm{a}^l_i}{\partial \mathrm{h}^l_i}\sum_j \frac{\partial^2 E}{\partial \mathrm{h}^{l+1}_j \partial \mathrm{h}^{l+1}_k}\frac{\partial \mathrm{h}^{l+1}_j}{\partial \mathrm{h}^l_m} + \frac{\partial E}{\partial \mathrm{h}^{l+1}_k}\delta_{i,m}\frac{\partial^2 \mathrm{a}^l_i}{\partial \mathrm{h}^{l\,2}_i}\right) \\
&= \delta_{i,m}\frac{\partial^2 \mathrm{a}^l_i}{\partial \mathrm{h}^{l\,2}_i}\left(\sum_k W^{l+1}_{k,i}\frac{\partial E}{\partial \mathrm{h}^{l+1}_k}\right) + \sum_{k,j} W^{l+1}_{k,i}\frac{\partial \mathrm{a}^l_i}{\partial \mathrm{h}^l_i}\frac{\partial^2 E}{\partial \mathrm{h}^{l+1}_j \partial \mathrm{h}^{l+1}_k}W^{l+1}_{j,m}\frac{\partial \mathrm{a}^l_m}{\partial \mathrm{h}^l_m} \\
&= \delta_{i,m}\frac{\partial^2 \mathrm{a}^l_i}{\partial \mathrm{h}^{l\,2}_i}\frac{\partial E}{\partial \mathrm{a}^l_i} + \sum_{k,j} W^{l+1}_{k,i}\frac{\partial \mathrm{a}^l_i}{\partial \mathrm{h}^l_i}\frac{\partial^2 E}{\partial \mathrm{h}^{l+1}_j \partial \mathrm{h}^{l+1}_k}\frac{\partial \mathrm{a}^l_m}{\partial \mathrm{h}^l_m}W^{l+1}_{j,m}
\end{aligned}$$

and defining

$$B^l_{i,j} = \delta_{i,j} \frac{\partial \mathrm{a}^l_i}{\partial \mathrm{h}^l_i} = \delta_{i,j} f'(\mathrm{h}^l_i) \tag{A.9}$$

$$D^l_{i,j} = \delta_{i,j} \frac{\partial^2 \mathrm{a}^l_i}{\partial \mathrm{h}^{l\,2}_i} \frac{\partial E}{\partial \mathrm{a}^l_i} = \delta_{i,j} f''(\mathrm{h}^l_i) \frac{\partial E}{\partial \mathrm{a}^l_i} \tag{A.10}$$

yields Eq. A.6.

For further details and on how to calculate the diagonal blocks of the Gauss-Newton and Fisher matrix, we refer the reader to [29] and [229].

Appendix B

Bayesian binary neural network PyTorch code

We provide a basic PyTorch implementation with classes corresponding to `nn.Linear` in Listings B.1 to B.3. Listing B.1 provides a binary base class implementing the forward pass, reusing the `.weights` and `.bias` attribute of the base class as the logit parameters of the elementwise Bernoulli distributions in the approximate posterior. The forward pass uses the local reparameterisation trick when training, such that it is differentiable, and samples the weights as -1 or 1 when testing.

The classes in Listings B.2 and B.3 inherit from this class, adding a `kl_divergence` method that calculates the corresponding divergence to the prior. Note that for continual learning the classes would have to slightly modified: rather than just storing the prior parameter as a scalar (as it is shared across all weights in the supervised case) it would be necessary to use the `.register_buffer` method to store a tensor of the same size as the `.weight` and `.bias` attributes. After having finished training on one dataset, the data in these prior tensor would then be overwritten by the current values of `.weight` and `.bias`. We use that $\log \sigma(x) = -\zeta(-x)$, where $\sigma(x) = (1+e^{-x})^{-1}$ is the sigmoid and $\zeta(x) = \log(1+e^{x})$ the softplus function. This allows us to calculate log probabilities directly as a function of the logits in a numerically stable way without explicitly transforming the logits into probability space.

Listing B.1: Linear base class that implements the forward pass.

```
def bin_mean_var(logits):
    if logits is None:
        return  None, None
    probs = logits.sigmoid()
    mean = 2 * probs - 1
    var = 4 * probs * (1 - probs)
    return mean, var

def bin_sample(logits):
    if logits is None:
        return None
    d = dist.Bernoulli(logits=logits)
    return 2 * d.sample() - 1

class BinaryLinear(nn.Linear):
    def forward(self, x):
        if self.training:
            m_w, v_w = bin_mean_var(
                self.weights)
            m_b, v_b = bin_mean_var(
                self.bias)
            m_a = F.linear(
                x, m_w, m_b)
            sd_a = F.linear(
                x.pow(2),
                v_w, v_b).sqrt()
            d = dist.Normal(m_a, sd_a)
            return d.rsample()
        else:
            w = bin_sample(self.weights)
            b = bin_sample(self.bias)
            return F.linear(x, w, b)
```

Listing B.2: Linear binary layer with Bernoulli prior on the weights.

```
def bernoulli_kl(logits1, logits2):
    p = logits.sigmoid()
    # 1 - sigmoid(x) = sigmoid(-x)
    one_m_p = logits.neg().sigmoid()
    t1 = F.softplus(-logits2)
    t2 = F.softplus(-logits1)
    t3 = F.softplus(logits2)
    t4 = F.softplus(logits1)
    return (p * (t1 - t2) +
        one_m_p * (t3 - t4))

class BernoulliLinear(BinaryLinear):
    def __init__(self, *args,
                 p_logits=0.,
                 **kwargs):
        super().__init__(
            *args, **kwargs)
        self.p_logits = p_logits

    def kl_divergence(self):
        w_p_logits = torch.full_like(
            self.weights,
            self.p_logits)
        w_kl = bernoulli_kl(
            self.weights, w_p_logits)
        if self.bias is None:
            return w_kl.sum()
        b_p_logits = torch.full_like(
            self.bias, self.p_logits)
        b_kl = bernoulli_kl(
            self.bias, b_p_logits)
        return w_kl.sum() + b_kl.sum()
```

Listing B.3: Linear binary layer with Beta-Bernoulli prior on the weights

```
def betab_neg_log_prob(logits, delta):
    t1 = F.softplus(-logits)
    t2 = F.softplus(logits)
    return delta * (t1 + t2)

class BetaLinear(BinaryLinear):
    def __init__(self, *args, delta=0., **kwargs):
        super().__init__(*args, **kwargs)
        self.delta = delta

    def kl_divergence(self):
        delta = torch.full_like(self.weights, self.delta)
        neg_log_prob = betab_neg_log_prob(self.weights, delta).sum()
        if self.bias is not None:
            delta = torch.full_like(self.bias, self.delta)
            neg_log_prob += betab_neg_log_prob(self.bias, delta).sum()
        return neg_log_prob
```

Appendix C

The extended Matheron's rule to matrix normal distributions

The original Matheron's rule [156, 137, 59] for sampling conditional Gaussian variables states the following. If the joint multivariate Gaussian distribution is

$$\begin{pmatrix} \mathrm{vec}(\boldsymbol{W}) \\ \mathrm{vec}(\boldsymbol{U}) \end{pmatrix} \sim p(\mathrm{vec}(\boldsymbol{W}), \mathrm{vec}(\boldsymbol{U})) := \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}),$$
$$\boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{\boldsymbol{WW}} & \boldsymbol{\Sigma}_{\boldsymbol{WU}} \\ \boldsymbol{\Sigma}_{\boldsymbol{UW}} & \boldsymbol{\Sigma}_{\boldsymbol{UU}} \end{pmatrix},$$

then, conditioned on $\boldsymbol{U}$, sampling $\boldsymbol{W} \sim p(\mathrm{vec}(\boldsymbol{W}), \mathrm{vec}(\boldsymbol{U}))$ can be done as

$$\mathrm{vec}(\boldsymbol{W}) = \mathrm{vec}(\bar{\boldsymbol{W}}) + \boldsymbol{\Sigma}_{\boldsymbol{WU}} \boldsymbol{\Sigma}_{\boldsymbol{UU}}^{-1} (\mathrm{vec}(\boldsymbol{U}) - \mathrm{vec}(\bar{\boldsymbol{U}})),$$
$$\mathrm{vec}(\bar{\boldsymbol{W}}), \mathrm{vec}(\bar{\boldsymbol{U}}) \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}).$$

Matheron's rule can provide significant speed-ups if $\mathrm{vec}(\boldsymbol{U})$ has significantly smaller dimensions than that of $\mathrm{vec}(\boldsymbol{W})$, and the Cholesky decomposition of $\boldsymbol{\Sigma}$ can be computed with low costs (e.g. due to the specific structure in $\boldsymbol{\Sigma}$). Recall from the main text that the augmented prior is

$$p(\mathrm{vec}(\boldsymbol{W}), \mathrm{vec}(\boldsymbol{U})) = \mathcal{N}\left(\mathbf{0}, \begin{pmatrix} \sigma_c^2 \boldsymbol{I} \otimes \sigma_r^2 \boldsymbol{I} & \sigma_c \boldsymbol{Z}_c^\top \otimes \sigma_r \boldsymbol{Z}_r^\top \\ \sigma_c \boldsymbol{Z}_c \otimes \sigma_r \boldsymbol{Z}_r & \boldsymbol{\Psi}_c \otimes \boldsymbol{\Psi}_r \end{pmatrix}\right), \tag{C.1}$$

and the corresponding conditional distribution is:

$$p(\mathrm{vec}(\boldsymbol{W})|\,\mathrm{vec}(\boldsymbol{U})) = \mathcal{N}(\sigma_c\sigma_r\,\mathrm{vec}(\boldsymbol{Z}_r\boldsymbol{\Psi}_r^{-1}\boldsymbol{U}\boldsymbol{\Psi}_c^{-1}\boldsymbol{Z}_c^\top), \sigma_c^2\sigma_r^2(\boldsymbol{I}-\boldsymbol{Z}_c^\top\boldsymbol{\Psi}_c^{-1}\boldsymbol{Z}_c\otimes\boldsymbol{Z}_r^\top\boldsymbol{\Psi}_r^{-1}\boldsymbol{Z}_r)). \tag{C.2}$$

Therefore, while $\dim(\mathrm{vec}(\boldsymbol{U}))$ is indeed significantly smaller than of $\dim(\mathrm{vec}(\boldsymbol{W}))$ by construction, the joint covariance matrix does not support fast Cholesky decompositions, meaning that Matheron's rule for efficient sampling does not directly apply here.

However, in the full augmented space, the joint distribution does have an efficient matrix normal form: $p(\boldsymbol{W},\boldsymbol{U}_c,\boldsymbol{U}_r,\boldsymbol{U}_c) = \mathcal{MN}(0,\boldsymbol{\Sigma}_r,\boldsymbol{\Sigma}_c)$. Furthermore, the row and column covariance matrices $\boldsymbol{\Sigma}_r$ and $\boldsymbol{\Sigma}_c$ are parameterised by their Cholesky decompositions, meaning that sampling from the joint distribution $p(\boldsymbol{W},\boldsymbol{U}_c,\boldsymbol{U}_r,\boldsymbol{U})$ can be done in a fast way. Importantly, Cholesky decompositions for $p(\boldsymbol{U})$'s row and column covariance matrices $\boldsymbol{\Psi}_r$ and $\boldsymbol{\Psi}_c$ can be computed in $\mathcal{O}(M_{out}^3)$ and $\mathcal{O}(M_{in}^3)$ time, respectively, which are much faster than the multi-variate Gaussian case that requires $\mathcal{O}(M_{in}^3M_{out}^3)$ time. Observing these, we extend Matheron's rule to sample $p(\boldsymbol{W}|\boldsymbol{U})$ where $p(\boldsymbol{W},\boldsymbol{U})$ is the marginal distribution of $p(\boldsymbol{W},\boldsymbol{U}_c,\boldsymbol{U}_r,\boldsymbol{U}_c) = \mathcal{MN}(\boldsymbol{0},\boldsymbol{\Sigma}_r,\boldsymbol{\Sigma}_c)$.

In detail, for drawing a sample from $p(\boldsymbol{W}|\boldsymbol{U})$ we need to draw a sample from the joint $p(\boldsymbol{W},\boldsymbol{U})$. To do so, we sample from the augmented prior $\bar{\boldsymbol{W}},\bar{\boldsymbol{U}}_c,\bar{\boldsymbol{U}}_r,\bar{\boldsymbol{U}} \sim p(\bar{\boldsymbol{W}},\bar{\boldsymbol{U}}_c,\bar{\boldsymbol{U}}_r,\bar{\boldsymbol{U}}) = \mathcal{MN}(0,\boldsymbol{\Sigma}_r,\boldsymbol{\Sigma}_c)$, computed using the Cholesky decompositions of $\boldsymbol{\Sigma}_r$ and $\boldsymbol{\Sigma}_c$

$$\begin{pmatrix}\bar{\boldsymbol{W}} & \bar{\boldsymbol{U}}_c\\ \bar{\boldsymbol{U}}_r & \bar{\boldsymbol{U}}\end{pmatrix} = \begin{pmatrix}\sigma_r\boldsymbol{I} & \boldsymbol{0}\\ \boldsymbol{Z}_r & \boldsymbol{D}_r\end{pmatrix}\begin{pmatrix}E_1 & E_2\\ E_3 & E_4\end{pmatrix}\begin{pmatrix}\sigma_c\boldsymbol{I} & \boldsymbol{Z}_c^\top\\ \boldsymbol{0} & \boldsymbol{D}_c\end{pmatrix},$$

where $E_1 \in \mathbb{R}^{d_{out}\times d_{in}}$, $E_2 \in \mathbb{R}^{d_{out}\times M_{in}}$, $E_3 \in \mathbb{R}^{M_{out}\times d_{in}}$, $E_4 \in \mathbb{R}^{M_{out}\times M_{in}}$ are standard Gaussian noise samples, and $\bar{\boldsymbol{W}} \in \mathbb{R}^{d_{out}\times d_{in}}$ and $\bar{\boldsymbol{U}} \in \mathbb{R}^{M_{out}\times M_{in}}$. Then we construct the conditional sample $\boldsymbol{W} \sim p(\boldsymbol{W}|\boldsymbol{U})$ as follows, similar to Matheron's rule in the multivariate Gaussian case

$$\boldsymbol{W} = \bar{\boldsymbol{W}} + \sigma_r\sigma_c\boldsymbol{Z}_r^\top\boldsymbol{\Psi}_r^{-1}(\boldsymbol{U}-\bar{\boldsymbol{U}})\boldsymbol{\Psi}_c^{-1}\boldsymbol{Z}_c. \tag{C.3}$$

From the above equations we see that $\bar{\boldsymbol{U}}_r$ and $\bar{\boldsymbol{U}}_c$ do not contribute to the final $\bar{\boldsymbol{W}}$ sample. Therefore we do not need to compute $\bar{\boldsymbol{U}}_r$ and $\bar{\boldsymbol{U}}_c$, and we write the separate expressions for $\bar{\boldsymbol{W}}$ and $\bar{\boldsymbol{U}}$ as

$$\bar{\boldsymbol{W}} = \sigma_r \sigma_c E_1, \quad \bar{\boldsymbol{U}} = \underbrace{\boldsymbol{Z}_r E_1 \boldsymbol{Z}_c^\top}_{\bar{\boldsymbol{U}}_1} + \underbrace{\boldsymbol{Z}_r E_2 \boldsymbol{D}_c}_{\bar{\boldsymbol{U}}_2} + \underbrace{\boldsymbol{D}_r E_3 \boldsymbol{Z}_c^\top}_{\bar{\boldsymbol{U}}_3} + \underbrace{\boldsymbol{D}_r E_4 \boldsymbol{D}_c}_{\bar{\boldsymbol{U}}_4}. \tag{C.4}$$

Note that $\bar{\boldsymbol{U}}$ is a sum of four samples from matrix normal distributions. In particular, we have that

$$\bar{\boldsymbol{U}}_2 \overset{d}{\sim} \mathcal{MN}(\boldsymbol{0}, \boldsymbol{Z}_r \boldsymbol{Z}_r^\top, \boldsymbol{D}_c^2) \quad \text{and} \quad \bar{\boldsymbol{U}}_3 \overset{d}{\sim} \mathcal{MN}(\boldsymbol{0}, \boldsymbol{D}_r^2, \boldsymbol{Z}_c \boldsymbol{Z}_c^\top).$$

Hence instead of sampling the "long and thin" Gaussian noise matrices E_2 and E_3, we can reduce variance by sampling standard Gaussian noise matrices $\tilde{E}_2, \tilde{E}_3 \in \mathbb{R}^{M_{out} \times M_{in}}$, and calculate $\bar{\boldsymbol{U}}$ as

$$\bar{\boldsymbol{U}} = \boldsymbol{Z}_r E_1 \boldsymbol{Z}_c^\top + \hat{L}_r \tilde{E}_2 \boldsymbol{D}_c + \boldsymbol{D}_r \tilde{E}_3 \hat{L}_c^\top + \boldsymbol{D}_r E_4 \boldsymbol{D}_c. \tag{C.5}$$

This is enabled by calculating the Cholesky decompositions $\hat{L}_r \hat{L}_r^\top = \boldsymbol{Z}_r \boldsymbol{Z}_r^\top$ and $\hat{L}_c \hat{L}_c^\top = \boldsymbol{Z}_c \boldsymbol{Z}_c^\top$, which have $\mathcal{O}(M_{out}^3)$ and $\mathcal{O}(M_{in}^3)$ run-time costs, respectively. As a reminder, the Cholesky factors are *square* matrices, i.e. $\hat{L}_r \in \mathbb{R}^{M_{out} \times M_{out}}$, $\hat{L}_c \in \mathbb{R}^{M_{in} \times M_{in}}$). We name the approach the *extended Matheron's rule* for sampling conditional Gaussians when the full joint has a matrix normal form.

As to verify the proposed approach, we compute the mean and the variance of the random variable $\bar{\boldsymbol{W}}$ defined in Eq. C.3, and check if they match the mean and variance of Eq. C.2. First as $\bar{\boldsymbol{W}}, \bar{\boldsymbol{U}}$ have zero mean, it is straightforward to verify that $\mathbb{E}[W] = \sigma_r \sigma_c \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \boldsymbol{U} \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c$ which matches the mean of Eq. C.2. For the variance

of $\mathrm{vec}(\boldsymbol{W})$, it requires computing the following terms

$$\begin{aligned}\mathbb{V}(\mathrm{vec}(\boldsymbol{W})) =& \mathbb{V}(\mathrm{vec}(\bar{\boldsymbol{W}})) + \mathbb{V}(\mathrm{vec}(\sigma_r\sigma_c \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \bar{\boldsymbol{U}} \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c)) \\ & - 2\mathrm{Cov}[\mathrm{vec}(\boldsymbol{W}), \mathrm{vec}(\sigma_r\sigma_c \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \bar{\boldsymbol{U}} \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c)] \\ =:& \boldsymbol{A}_1 + \boldsymbol{A}_2 - 2\boldsymbol{A}_3. \end{aligned} \tag{C.6}$$

First it can be shown that

$$\begin{aligned}&\boldsymbol{A}_1 = \sigma_r^2\sigma_c^2 \boldsymbol{I} \quad \text{since } \bar{\boldsymbol{W}} \sim \mathcal{MN}(\boldsymbol{0}, \sigma_r^2\boldsymbol{I}, \sigma_c^2\boldsymbol{I}), \\ &\boldsymbol{A}_2 = \sigma_r^2\sigma_c^2 \boldsymbol{Z}_c^\top \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c \otimes \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \boldsymbol{Z}_r \\ &\text{as } \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \bar{\boldsymbol{U}} \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c \overset{d}{\sim} \mathcal{MN}(\boldsymbol{0}, \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \boldsymbol{Z}_r, \boldsymbol{Z}_c^\top \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c).\end{aligned}$$

For the correlation term $\boldsymbol{A}_3$, we notice that $\bar{\boldsymbol{W}}$ and $\bar{\boldsymbol{U}}$ only share the noise matrix $\boldsymbol{E}_1$ in the joint sampling procedure Eq. C.4. This also means

$$\begin{aligned}\boldsymbol{A}_3 &= \sigma_r^2\sigma_c^2 \mathrm{Cov}[\mathrm{vec}(\boldsymbol{E}_1), \mathrm{vec}(\boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \boldsymbol{Z}_r \boldsymbol{E}_1 \boldsymbol{Z}_c^\top \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c)] \\ &= \sigma_r^2\sigma_c^2 \boldsymbol{Z}_c^\top \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c \otimes \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \boldsymbol{Z}_r.\end{aligned}$$

Plugging in $\boldsymbol{A}_1, \boldsymbol{A}_2, \boldsymbol{A}_3$ into Eq. C.6 verifies that $\mathbb{V}(\mathrm{vec}(\boldsymbol{W}))$ matches the variance of the conditional distribution $p(\mathrm{vec}(\boldsymbol{W}) | \mathrm{vec}(\boldsymbol{U}))$, showing that the proposed extended Matheron's rule indeed draws samples from the conditional distribution.

As for sampling $\boldsymbol{W}$ from $q(\boldsymbol{W}|\boldsymbol{U})$, since it has the same mean but a rescaled covariance as compared with $p(\boldsymbol{W}|\boldsymbol{U})$, we can compute the samples by adapting the extend Matheron's rule as follows. Notice that the mean of $\boldsymbol{W}$ in Eq. C.3 is $\mathbb{E}[\boldsymbol{W}|\boldsymbol{U}] = \sigma_r\sigma_c \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \boldsymbol{U} \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c$, therefore by rearranging terms, Eq. C.3 can be re-written as

$$\begin{aligned}\boldsymbol{W} =& \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \boldsymbol{U} \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c + [\bar{\boldsymbol{W}} - \sigma_r\sigma_c \boldsymbol{Z}_r^\top \boldsymbol{\Psi}_r^{-1} \bar{\boldsymbol{U}} \boldsymbol{\Psi}_c^{-1} \boldsymbol{Z}_c] \\ :=& \text{mean} + \text{noise}.\end{aligned}$$

So sampling from $q(\boldsymbol{W}|\boldsymbol{U})$ can be done by rescaling the noise term in the above

equation with the scale parameter λ. In summary, the extended Matheron's rule for sampling $q(\boldsymbol{W}|\boldsymbol{U})$ is as follows

$$\begin{aligned} \boldsymbol{W} &= \lambda\bar{\boldsymbol{W}} + \sigma_r\sigma_c\boldsymbol{Z}_r^\top\boldsymbol{\Psi}_r^{-1}(\boldsymbol{U} - \lambda\bar{\boldsymbol{U}})\boldsymbol{\Psi}_c^{-1}\boldsymbol{Z}_c, \\ \bar{\boldsymbol{W}}, \bar{\boldsymbol{U}} &\sim p(\bar{\boldsymbol{W}}, \bar{\boldsymbol{U}}_c, \bar{\boldsymbol{U}}_r, \bar{\boldsymbol{U}}). \end{aligned} \tag{C.7}$$

Plugging in $\sigma_r\sigma_c = \sigma$ here returns the conditional sampling rule Eq. 4.33 in the main text.

Appendix D

Additional results and tables

D.1 Vision pentathlon numerical results

Table D.1: Per dataset test accuracy at the end of training on the suite of vision datasets. SI is Synaptic Intelligence [371] and EWC Elastic Weight Consolidation [170]. We abbreviate Per-Task Laplace (one penalty per task) as PTL, Approximate Laplace (Laplace approximation of the full posterior at the mode of the approximate objective) and our Online Laplace approximation as OL. nMNIST refers to notMNIST, fMNIST to FashionMNIST and C10 to CIFAR10.

		Test Error (%)					
Method	Approximation	MNIST	nMNIST	fMNIST	SVHN	C10	Avg.
SI	n/a	87.27	79.12	84.61	77.44	57.61	77.21
PTL	Diagonal (EWC)	97.83	94.73	89.13	79.80	53.29	82.96
	Kronecker factored	97.85	94.92	89.31	85.75	58.78	85.32
AL	Diagonal	96.56	92.33	89.27	78.00	56.57	82.55
	Kronecker factored	97.90	94.88	90.08	85.24	58.63	85.35
OL	Diagonal	96.48	93.41	88.09	81.79	53.80	82.71
	Kronecker factored	97.17	94.78	90.36	85.59	59.11	85.40

D.2 Performance of the binary BNNs for different number of samples

Table D.2: Bayesian binary NN numerical results (4 samples).

	MNIST			CIFAR10		
	NLL	Accuracy (%)	Brier score	NLL	Accuracy (%)	Brier score
STE	0.027 ± 0.001	99.21 ± 0.02	0.013 ± 0.0	0.371 ± 0.004	92.84 ± 0.06	0.12 ± 0.001
Ensemble (STE)	0.023 ± 0.001	99.23 ± 0.04	0.012 ± 0.0	0.21 ± 0.001	94.31 ± 0.06	0.087 ± 0.0
Bernoulli (τ=1)	0.238 ± 0.001	95.04 ± 0.04	0.096 ± 0.001	1.161 ± 0.013	61.2 ± 0.55	0.538 ± 0.006
Bernoulli (τ=10^{-1})	0.088 ± 0.001	97.78 ± 0.04	0.038 ± 0.001	0.687 ± 0.004	78.86 ± 0.23	0.324 ± 0.002
Bernoulli (τ=10^{-2})	0.033 ± 0.001	98.97 ± 0.03	0.016 ± 0.0	0.348 ± 0.002	89.19 ± 0.1	0.165 ± 0.001
Bernoulli (τ=10^{-3})	0.032 ± 0.001	98.95 ± 0.02	0.016 ± 0.0	0.304 ± 0.001	91.3 ± 0.1	0.132 ± 0.001
Beta-Bernoulli (Δ=0)	0.03 ± 0.0	99.0 ± 0.01	0.015 ± 0.0	0.304 ± 0.003	91.59 ± 0.11	0.128 ± 0.001
Beta-Bernoulli (Δ=10^{-1})	0.115 ± 0.001	97.33 ± 0.03	0.047 ± 0.0	1.144 ± 0.014	61.46 ± 0.73	0.526 ± 0.006
Beta-Bernoulli (Δ=10^{-2})	0.048 ± 0.001	98.72 ± 0.03	0.021 ± 0.0	0.5 ± 0.004	84.92 ± 0.17	0.237 ± 0.002
Beta-Bernoulli (Δ=10^{-3})	0.029 ± 0.0	99.03 ± 0.02	0.015 ± 0.0	0.318 ± 0.002	90.28 ± 0.11	0.146 ± 0.001
Beta-Bernoulli (Δ=10^{-4})	0.029 ± 0.0	99.03 ± 0.02	0.015 ± 0.0	0.303 ± 0.001	91.43 ± 0.06	0.131 ± 0.0

Table D.3: Bayesian binary NN numerical results (16 samples).

	MNIST			CIFAR10		
	NLL	Accuracy (%)	Brier score	NLL	Accuracy (%)	Brier score
STE	0.027 ± 0.001	99.21 ± 0.02	0.013 ± 0.0	0.371 ± 0.004	92.84 ± 0.06	0.12 ± 0.001
Ensemble (STE)	$0.022 \pm nan$	$99.27 \pm nan$	$0.011 \pm nan$	$0.174 \pm nan$	$94.62 \pm nan$	$0.08 \pm nan$
Bernoulli (τ=1)	0.234 ± 0.001	95.35 ± 0.05	0.092 ± 0.001	1.08 ± 0.013	67.26 ± 0.56	0.502 ± 0.006
Bernoulli (τ=10^{-1})	0.086 ± 0.001	97.9 ± 0.02	0.036 ± 0.0	0.619 ± 0.004	83.87 ± 0.17	0.286 ± 0.002
Bernoulli (τ=10^{-2})	0.032 ± 0.0	99.02 ± 0.01	0.016 ± 0.0	0.3 ± 0.002	90.95 ± 0.12	0.143 ± 0.001
Bernoulli (τ=10^{-3})	0.032 ± 0.001	98.96 ± 0.02	0.016 ± 0.0	0.254 ± 0.001	91.95 ± 0.06	0.119 ± 0.001
Beta-Bernoulli (Δ=0)	0.029 ± 0.0	99.02 ± 0.01	0.015 ± 0.0	0.256 ± 0.002	92.08 ± 0.1	0.117 ± 0.001
Beta-Bernoulli (Δ=10^{-1})	0.114 ± 0.001	97.47 ± 0.02	0.046 ± 0.0	1.02 ± 0.011	68.42 ± 0.8	0.475 ± 0.005
Beta-Bernoulli (Δ=10^{-2})	0.047 ± 0.001	98.75 ± 0.02	0.021 ± 0.0	0.44 ± 0.002	88.37 ± 0.12	0.204 ± 0.001
Beta-Bernoulli (Δ=10^{-3})	0.028 ± 0.0	99.02 ± 0.02	0.014 ± 0.0	0.268 ± 0.001	91.62 ± 0.06	0.128 ± 0.001
Beta-Bernoulli (Δ=10^{-4})	0.028 ± 0.0	99.05 ± 0.01	0.014 ± 0.0	0.253 ± 0.001	92.08 ± 0.07	0.118 ± 0.0

D.3 Inducing weight numerical results

In Tables D.4 to D.7 we report the numerical results for Fig. 4.11.

Table D.4: Corrupted CIFAR-10 accuracy (↑) values (in %).

Method	Skew Intensity 1	2	3	4	5
Deterministic	87.90±2.31	82.02±2.84	76.31±3.80	68.91±4.81	57.94±5.10
Ensemble-W	89.45±2.27	83.94±2.71	78.40±3.61	71.18±4.53	60.15±4.82
FFG-W	83.80±2.43	76.22±3.10	69.30±4.11	61.82±4.66	50.72±4.68
FFG-U	86.90±2.47	80.33±3.14	74.34±4.06	67.23±4.75	57.00±4.83
Ensemble-U	87.35±2.39	80.45±3.19	73.89±4.23	66.52±4.96	54.89±5.26

Table D.5: Corrupted CIFAR-10 ECE (↓) values (in %).

Method	Skew Intensity 1	2	3	4	5
Deterministic	10.41±2.03	15.58±2.52	20.56±3.37	27.06±4.23	37.26±4.65
Ensemble-W	4.12±1.31	7.01±1.64	10.10±2.33	14.12±2.84	20.48±2.95
FFG-W	13.05±0.64	12.14±0.89	11.56±0.93	10.77±1.05	10.86±1.31
FFG-U	2.47±1.04	4.93±1.66	7.77±2.44	11.27±2.81	16.16±2.92
Ensemble-U	2.77±1.04	5.86±1.69	9.06±2.40	12.54±2.66	19.66±3.12

Table D.6: Corrupted CIFAR-100 accuracy (↑) values (in %).

Method	Skew Intensity 1	2	3	4	5
Deterministic	63.18±3.06	54.23±3.67	48.47±4.27	41.84±4.59	31.96±3.96
Ensemble-W	67.10±3.19	57.92±3.82	51.83±4.50	45.16±4.91	34.93±4.27
FFG-W	57.49±3.17	47.62±3.64	41.99±4.15	35.61±4.24	26.59±3.66
FFG-U	61.71±3.35	52.61±3.84	47.08±4.36	40.56±4.54	30.88±3.93
Ensemble-U	61.87±3.36	52.69±3.97	46.96±4.52	40.59±4.72	30.85±3.98

Table D.7: Corrupted CIFAR-100 ECE (↓) values (in %).

Method	Skew Intensity 1	2	3	4	5
Deterministic	30.06±2.70	37.50±3.17	42.48±3.66	48.41±4.06	57.19±3.60
Ensemble-W	12.31±2.04	17.03±2.31	20.36±2.61	24.16±2.98	29.72±2.49
FFG-W	14.28±0.78	11.07±1.11	11.13±0.85	11.21±1.29	11.96±1.68
FFG-U	4.64±1.66	7.80±2.03	10.42±2.39	13.98±2.83	19.17±2.72
Ensemble-U	5.84±1.91	10.28±2.43	13.60±2.96	17.54±3.42	23.56±3.16

In Tables D.8 to D.11 we report the corresponding results for pruning FFG-W and FFG-U. See Fig. 4.13 for visualisation.

Table D.8: Corrupted CIFAR-10 accuracy (↑) values (in %) for pruning FFG-W and FFG-U.

Method	Skew Intensity 1	2	3	4	5
FFG-W (100%)	83.80±2.43	76.22±3.10	69.30±4.11	61.82±4.66	50.72±4.68
FFG-W (50%)	83.39±2.66	75.58±3.23	68.43±4.21	60.69±4.75	49.75±4.73
FFG-W (10%)	84.04±2.61	76.21±3.24	69.35±4.18	61.65±4.72	50.51±4.74
FFG-W (1%)	84.16±2.31	76.48±3.02	69.72±3.92	62.72±4.47	51.26±4.58
FFG-W (0.1%)	46.33±1.30	41.92±1.44	38.91±1.59	36.03±1.76	32.43±1.88
FFG-U (100%)	86.90±2.47	80.33±3.14	74.34±4.06	67.23±4.75	57.00±4.83
FFG-U (75%)	86.99±2.37	80.64±2.95	74.99±3.81	67.87±4.53	57.33±4.65
FFG-U (50%)	86.93±2.36	80.70±2.93	75.10±3.76	68.14±4.44	57.66±4.49
FFG-U (25%)	85.93±2.39	79.41±2.96	73.48±3.82	66.52±4.52	55.57±4.59

Table D.9: Corrupted CIFAR-10 ECE (↓) values (in %) for pruning FFG-W and FFG-U.

Method	Skew Intensity 1	2	3	4	5
FFG-W (100%)	13.05±0.64	12.14±0.89	11.56±0.93	10.77±1.05	10.86±1.31
FFG-W (50%)	14.27±0.59	13.19±0.88	12.36±0.94	11.59±1.07	11.43±1.33
FFG-W (10%)	12.50±0.52	11.66±0.76	11.33±0.90	10.86±1.04	11.28±1.38
FFG-W (1%)	9.86±0.49	9.17±0.62	8.85±0.87	8.87±0.93	11.10±1.45
FFG-W (0.1%)	10.08±0.87	7.82±0.96	6.74±0.81	7.34±0.79	8.77±0.93
FFG-U (100%)	2.47±1.04	4.93±1.66	7.77±2.44	11.27±2.81	16.16±2.92
FFG-U (75%)	2.79±0.60	3.92±0.92	5.27±1.59	7.63±2.00	11.80±2.39
FFG-U (50%)	3.11±0.59	4.26±0.92	5.44±1.58	7.58±1.94	11.38±2.26
FFG-U (25%)	5.27±0.35	5.33±0.57	6.20±1.19	7.95±1.55	11.19±2.21

Table D.10: Corrupted CIFAR-100 accuracy (↑) values (in %) for pruning FFG-W and FFG-U.

Method	Skew Intensity 1	2	3	4	5
FFG-W (100%)	57.49±3.17	47.62±3.64	41.99±4.15	35.61±4.24	26.59±3.66
FFG-W (50%)	57.16±3.16	47.33±3.65	41.43±4.18	35.02±4.25	25.89±3.58
FFG-W (10%)	58.77±3.18	48.61±3.69	42.89±4.25	36.33±4.35	26.97±3.70
FFG-W (1%)	50.64±2.89	41.70±3.35	37.35±3.69	31.56±3.65	23.84±3.05
FFG-W (0.1%)	6.72±0.23	6.00±0.24	5.77±0.28	5.43±0.33	4.82±0.33
FFG-U (100%)	61.71±3.35	52.61±3.84	47.08±4.36	40.56±4.54	30.88±3.93
FFG-U (75%)	61.47±3.41	52.25±3.95	46.84±4.46	40.37±4.65	30.48±3.97
FFG-U (50%)	60.76±3.46	51.68±3.96	46.28±4.43	39.84±4.61	29.85±3.92
FFG-U (25%)	58.11±3.20	49.06±3.64	43.62±4.09	37.20±4.23	27.93±3.60

Table D.11: Corrupted CIFAR-100 ECE (↓) values (in %) for pruning FFG-W and FFG-U.

Method	Skew Intensity 1	2	3	4	5
FFG-W (100%)	14.28±0.78	11.07±1.11	11.13±0.85	11.21±1.29	11.96±1.68
FFG-W (50%)	15.42±0.84	12.04±1.19	11.85±0.91	11.77±1.31	11.71±1.61
FFG-W (10%)	10.85±0.84	9.08±0.95	9.96±1.10	10.97±1.74	13.35±1.98
FFG-W (1%)	14.86±1.10	11.99±1.13	11.78±1.20	11.48±1.47	11.37±1.64
FFG-W (0.1%)	3.21±0.57	4.47±0.55	5.28±0.70	6.53±0.96	7.79±1.01
FFG-U (100%)	4.64±1.66	7.80±2.03	10.42±2.39	13.98±2.83	19.17±2.72
FFG-U (75%)	4.78±1.58	7.29±1.91	9.59±2.33	13.05±2.91	18.39±2.87
FFG-U (50%)	5.31±1.55	7.10±1.81	9.06±2.18	12.35±2.82	17.43±2.83
FFG-U (25%)	9.74±0.99	8.17±1.01	8.67±1.21	10.53±1.85	13.22±2.15

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016. 21, 145

[2] T. Adel, H. Zhao, and R. E. Turner. Continual learning with adaptive weights (CLAW). In *ICLR*, 2020. 93

[3] F. V. Agakov and D. Barber. An auxiliary variational method. In *ICONIP*, 2004. 118

[4] S. Ahn, A. Korattikara, and M. Welling. Bayesian posterior sampling via stochastic gradient Fisher scoring. In *ICML*, 2012. 36

[5] M. Alizadeh, J. Fernández-Marqués, N. D. Lane, and Y. Gal. A systematic study of binary neural networks' optimisation. In *ICLR*, 2019. 98, 100

[6] S.-I. Amari. Neural learning in structured parameter spaces - natural Riemannian gradient. In *NeurIPS*, 1997. 50

[7] S.-I. Amari. Natural gradient works efficiently in learning. *Neural Computation*, 1998. 50

[8] T. Anthony, Z. Tian, and D. Barber. Thinking fast and slow with deep learning and tree search. In *NeurIPS*, 2017. 14

[9] J. Antoran, J. U. Allingham, D. Janz, E. Daxberger, E. Nalisnick, and J. M. Hernández-Lobato. Linearised Laplace inference in networks with normalisation layers and the neural g-prior. In *AABI*, 2021. 73

[10] Y. Arjevani and M. Field. Analytic characterization of the hessian in shallow relu models: A tale of symmetry. *arXiv preprint arXiv:2008.01805*, 2020. 55

[11] J. T. Ash, S. Goel, A. Krishnamurthy, and S. Kakade. Gone fishing: Neural active learning with Fisher embeddings. *arXiv preprint arXiv:2106.09675*, 2021. 75

[12] A. Atanov, A. Ashukha, D. Molchanov, K. Neklyudov, and D. Vetrov. Uncertainty estimation via stochastic batch normalization. *arXiv preprint arXiv:1802.04893*, 2018. 37

[13] A. Atanov, A. Ashukha, K. Struminsky, D. Vetrov, and M. Welling. The deep weight prior. In *ICLR*, 2019. 139

[14] H. Attias. Inferring parameters and structure of latent variable models by variational Bayes. In *UAI*, 1999. 34

[15] J. Bae, G. Zhang, and R. Grosse. Eigenvalue corrected noisy natural gradient. *arXiv preprint arXiv:1811.12565*, 2018. 74

[16] A. K. Balan, V. Rathod, K. P. Murphy, and M. Welling. Bayesian dark knowledge. In *NeurIPS*, 2015. 59

[17] D. Barber. *Bayesian reasoning and machine learning*. Cambridge university press, 2012. 29

[18] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: A survey. *JMLR*, 2017. 100

[19] T. Bayes. An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, FRS communicated by Mr. Price, in a letter to John Canton, AMFR S. *Philosophical Transactions of the Royal Society of London*, 1763. 16

[20] A. Bekasov and I. Murray. Bayesian adversarial spheres: Bayesian inference and adversarial examples in a noiseless setting. *arXiv preprint arXiv:1811.12335*, 2018. 40

[21] Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013. 100

[22] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep universal probabilistic programming. *JMLR*, 2019. 19, 106, 130, 145, 146, 152

[23] T. Bird, J. Kunze, and D. Barber. Stochastic variational optimization. *arXiv preprint arXiv:1809.04855*, 2018. 100

[24] C. M. Bishop. *Pattern recognition and machine learning*. Springer, 2006. 29

[25] J. Bjorck, C. Gomes, B. Selman, and K. Q. Weinberger. Understanding batch normalization. In *NeurIPS*, 2018. 23

[26] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. Weight uncertainty in neural networks. In *ICML*, 2015. 17, 34, 66, 73, 93, 103, 114, 115, 116, 118, 125, 128, 129, 151

[27] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. 15

[28] A. Botev. *The Gauss-Newton matrix for Deep Learning models and its applications*. PhD thesis, UCL, 2020. 43

[29] A. Botev, H. Ritter, and D. Barber. Practical Gauss-Newton optimisation for deep learning. In *ICML*, 2017. 43, 49, 50, 60, 64, 76, 166, 168

[30] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*, 2010. 26, 100

[31] L. Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*. Springer, 2012. 26

[32] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula,

A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs. https://github.com/google/jax, 2018. 74

[33] J. Bradshaw, A. G. d. G. Matthews, and Z. Ghahramani. Adversarial examples, uncertainty, and transfer testing robustness in Gaussian process hybrid deep networks. *arXiv preprint arXiv:1707.02476*, 2017. 139

[34] G. W. Brier. Verification of forecasts expressed in terms of probability. *Monthly Weather Review*, 1950. 109

[35] J. Bröcker. Reliability, sufficiency, and the decomposition of proper scores. *QJRMAM*, 2009. 38

[36] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020. 116

[37] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020. 14

[38] W. Buntine and A. Weigend. Bayesian back-propagation. *Complex Systems*, 1991. 42

[39] L. Butyrev, G. Kontes, C. Löffler, and C. Mutschler. Overcoming catastrophic forgetting via Hessian-free curvature estimates. https://openreview.net/forum?id=H1ls_eSKPH, 2020. 93

[40] G. Carbone, M. Wicker, L. Laurenti, A. Patane, L. Bortolussi, and G. Sanguinetti. Robustness of Bayesian neural networks to gradient-based attacks. In *NeurIPS*, 2020. 40

[41] B. P. Carlin and S. Chib. Bayesian model choice via Markov chain Monte Carlo methods. *JRSS B*, 1995. 140

[42] R. Caruana. Multitask learning. *Machine learning*, 1997. 15

[43] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *ICCV*, 2015. 15

[44] T. Chen, E. Fox, and C. Guestrin. Stochastic gradient Hamiltonian Monte Carlo. In *ICML*, 2014. 36

[45] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015. 21

[46] F. Chollet et al. Keras, 2015. 146

[47] B. Coker, W. Pan, and F. Doshi-Velez. Wide mean-field variational Bayesian neural networks ignore the data. *arXiv preprint arXiv:2102.01936*, 2021. 42

[48] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NeurIPS*, 2015. 98, 100, 114

[49] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv preprint arXiv:1602.02830*, 2016. 98, 100, 114

[50] N. Cristianini, J. Shawe-Taylor, et al. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000. 20

[51] F. Dangel, F. Kunstner, and P. Hennig. Backpack: Packing more into backprop. In *ICLR*, 2020. 50, 51

[52] E. Daxberger, A. Kristiadi, A. Immer, R. Eschenhagen, M. Bauer, and P. Hennig. Laplace Redux–effortless Bayesian deep learning. In *NeurIPS*, 2021. 96

[53] E. Daxberger, E. Nalisnick, J. U. Allingham, J. Antorán, and J. M. Hernández-Lobato. Bayesian deep learning via subnetwork inference. In *ICML*, 2021. 75, 139

[54] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009. 52, 98

[55] J. S. Denker and Y. Lecun. Transforming neural-net output levels to probability distributions. In *NeurIPS*, 1991. 42

[56] A. Der Kiureghian and O. Ditlevsen. Aleatory or epistemic? Does it matter? *Structural safety*, 2009. 40

[57] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 14

[58] S. Dieleman, J. Schlüter, C. Raffel, E. Olson, S. K. Sønderby, D. Nouri, et al. Lasagne: First release., 2015. 43, 50, 62

[59] A. Doucet. A note on efficient conditional simulation of Gaussian distributions. *Technical report, University of British Columbia*, 2010. 126, 173

[60] F. Draxler, K. Veschgini, M. Salmhofer, and F. Hamprecht. Essentially no barriers in neural network energy landscape. In *ICML*, 2018. 55

[61] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 2011. 27

[62] M. W. Dusenberry, G. Jerfel, Y. Wen, Y.-a. Ma, J. Snoek, K. Heller, B. Lakshminarayanan, and D. Tran. Efficient and scalable Bayesian neural nets with rank-1 factors. In *ICML*, 2020. 116, 128, 138

[63] R. Eschenhagen, E. Daxberger, P. Hennig, and A. Kristiadi. Mixtures of Laplace approximations for improved post-hoc uncertainty in deep learning. *arXiv preprint arXiv:2111.03577*, 2021. 74, 164

[64] E. Eskin, A. J. Smola, and S. Vishwanathan. Laplace propagation. In *NeurIPS*, 2004. 93

[65] P. Esposito. BLiTZ - Bayesian layers in Torch zoo (a Bayesian deep learing library for Torch). https://github.com/piEsposito/blitz-bayesian-deep-learning/, 2020. 161

[66] S. Farquhar and Y. Gal. Towards robust evaluations of continual learning. *arXiv preprint arXiv:1805.09733*, 2018. 94

[67] S. Farquhar, M. Osborne, and Y. Gal. Radial Bayesian neural networks: Robust variational inference in big models. In *AISTATS*, 2020. 99

[68] S. Farquhar, L. Smith, and Y. Gal. Liberty or depth: Deep Bayesian neural nets do not need complex weight posterior approximations. In *NeurIPS*, 2020. 162

[69] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel, and D. Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017. 95

[70] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, 2017. 73

[71] R. A. Fisher. On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 1922. 50

[72] R. A. Fisher. Theory of statistical estimation. In *Mathematical Proceedings of the Cambridge Philosophical Society*, 1925. 50

[73] A. Foong, D. Burt, Y. Li, and R. E. Turner. On the expressiveness of approximate inference in Bayesian neural networks. In *NeurIPS*, 2020. 37, 42, 98

[74] A. Y. Foong, Y. Li, J. M. Hernández-Lobato, and R. E. Turner. 'in-between' uncertainty in Bayesian neural networks. *arXiv preprint arXiv:1906.11537*, 2019. 74, 129, 147

[75] F. D. Foresee and M. T. Hagan. Gauss-Newton approximation to Bayesian learning. In *ICNN*, 1997. 49

[76] V. Fortuin, A. Garriga-Alonso, S. W. Ober, F. Wenzel, G. Ratsch, R. E. Turner, M. van der Wilk, and L. Aitchison. Bayesian neural network priors revisited. In *ICLR*, 2022. 165

[77] R. M. French. Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 1999. 15, 76, 98

[78] C. G. Frye, J. Simon, N. S. Wadia, A. Ligeralde, M. R. DeWeese, and K. E. Bouchard. Critical point-finding methods reveal gradient-flat regions of deep network losses. *Neural Computation*, 2021. 55

[79] K. Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural Networks*, 1988. 22

[80] K. Fukushima and S. Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and Cooperation in Neural Nets*. Springer, 1982. 22

[81] Y. Gal. *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge, 2016. 41

[82] Y. Gal and Z. Ghahramani. Bayesian convolutional neural networks with Bernoulli approximate variational inference. *arXiv preprint arXiv:1506.02158*, 2015. 71

[83] Y. Gal and Z. Ghahramani. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *ICML*, 2016. 37, 62, 67, 109, 116

[84] Y. Gal and L. Smith. Sufficient conditions for idealised models to have no adversarial examples: a theoretical and empirical study with Bayesian neural networks. *arXiv preprint arXiv:1806.00667*, 2018. 40

[85] T. Garipov, P. Izmailov, D. Podoprikhin, D. Vetrov, and A. G. Wilson. Loss surfaces, mode connectivity, and fast ensembling of DNNs. In *NeurIPS*, 2018. 55

[86] T. George, C. Laurent, X. Bouthillier, N. Ballas, and P. Vincent. Fast approximate natural gradient descent in a Kronecker-factored eigenbasis. In *NeurIPS*, 2018. 74, 94

[87] Z. Ghahramani. Online variational Bayesian learning. http://www.gatsby.ucl.ac.uk/~zoubin/papers.html, 2000. Slides from talk presented at NeurIPS 2000 workshop on Online Learning. 76, 93

[88] Z. Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 2015. 29

[89] B. Ghorbani, S. Krishnan, and Y. Xiao. An investigation into neural net optimization via Hessian eigenvalue density. In *ICML*, 2019. 54, 55

[90] S. Ghosh, F. M. Delle Fave, and J. S. Yedidia. Assumed density filtering methods for learning Bayesian neural networks. In *AAAI*, 2016. 73

[91] S. Ghosh, J. Yao, and F. Doshi-Velez. Model selection in Bayesian neural networks via horseshoe priors. *JMLR*, 2019. 139

[92] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010. 21, 149

[93] T. Gneiting and A. E. Raftery. Strictly proper scoring rules, prediction, and estimation. *JASA*, 2007. 38, 39

[94] I. J. Goodfellow. Efficient per-example gradient computations. *arXiv preprint arXiv:1510.01799*, 2015. 51

[95] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013. 15, 76, 98, 113

[96] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. In *NeurIPS*, 2014. 14

[97] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014. 14, 40, 68

[98] I. J. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*. MIT Press, 2016. 14

[99] E. Grant, C. Finn, S. Levine, T. Darrell, and T. Griffiths. Recasting gradient-based meta-learning as hierarchical Bayes. In *ICLR*, 2018. 73

[100] A. Graves. Practical variational inference for neural networks. In *NeurIPS*, 2011. 17, 34, 66, 73, 135, 139

[101] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *ICASSP*, 2013. 14

[102] A. Graves, G. Wayne, and I. Danihelka. Neural Turing machines. *arXiv preprint arXiv:1410.5401*, 2014. 14

[103] R. Grosse and J. Martens. A Kronecker-factored approximate Fisher matrix for convolution layers. In *ICML*, 2016. 51, 52, 70, 91

[104] S. F. Gull. Developments in maximum entropy data analysis. In *Maximum entropy and Bayesian methods*. Springer, 1989. 42

[105] C. Guo, G. Pleiss, Y. Sun, and K. Weinberger. On calibration of modern neural networks. In *ICML*, 2017. 14, 38, 98

[106] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger. On calibration of modern neural networks. In *ICML*, 2017. 131

[107] A. K. Gupta and D. K. Nagar. *Matrix Variate Distributions*. CRC Press, 2018. 53, 56, 121

[108] R. Hadsell, D. Rao, A. A. Rusu, and R. Pascanu. Embracing change: Continual learning in deep neural networks. *Trends in Cognitive Sciences*, 2020. 16

[109] A. Hasanzadeh, E. Hajiramezanali, S. Boluki, M. Zhou, N. Duffield, K. Narayanan, and X. Qian. Bayesian graph neural networks with adaptive connection sampling. In *ICML*, 2020. 156

[110] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 1970. 35

[111] M. Havasi, R. Peharz, and J. M. Hernández-Lobato. Minimal random code learning: Getting bits back from compressed model parameters. In *ICLR*, 2019. 139

[112] H. He, G. Huang, and Y. Yuan. Asymmetric valleys: Beyond sharp and flat local minima. In *NeurIPS*, 2019. 55

[113] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *ICCV*, 2015. 21, 149

[114] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 14, 23, 44, 70, 116, 148, 153

[115] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *ECCV*, 2016. 14, 23, 44, 70, 116, 131, 148, 153

[116] X. He and H. Jaeger. Overcoming catastrophic interference using conceptor-aided backpropagation. In *ICLR*, 2018. 95

[117] J. Heek and N. Kalchbrenner. Bayesian inference for large scale image classification. In *ICLR*, 2020. 36

[118] M. Hein, M. Andriushchenko, and J. Bitterwolf. Why relu networks yield high-confidence predictions far away from the training data and how to mitigate the problem. In *CVPR*, 2019. 40

[119] K. Helwegen, J. Widdicombe, L. Geiger, Z. Liu, K.-T. Cheng, and R. Nusselder. Latent weights do not exist: Rethinking binarized neural network optimization. In *NeurIPS*, 2019. 114

[120] D. Hendrycks and T. Dietterich. Benchmarking neural network robustness to common corruptions and perturbations. *arXiv preprint arXiv:1903.12261*, 2019. 14, 39, 133, 135, 136

[121] D. Hendrycks and K. Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. In *ICLR*, 2017. 39

[122] J. Hensman, N. Fusi, and N. D. Lawrence. Gaussian processes for big data. In *UAI*, 2013. 94

[123] J.-M. Hernández-Lobato and R. Adams. Probabilistic backpropagation for scalable learning of Bayesian neural networks. In *ICML*, 2015. 111

[124] J. M. Hernández-Lobato and R. Adams. Probabilistic backpropagation for scalable learning of Bayesian neural networks. In *ICML*, 2015. 17, 62, 73

[125] T. Heskes. On "natural" learning and pruning in multilayered perceptrons. *Neural Computation*, 2000. 51

[126] N. J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002. 57

[127] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE SPM*, 2012. 14

[128] G. Hinton, N. Srivastava, and K. Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Coursera*, 2012. 27

[129] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 2006. 21

[130] G. E. Hinton and D. Van Camp. Keeping the neural networks simple by minimizing the description length of the weights. In *COLT*, 1993. 34, 73, 101

[131] M. Hobbhahn, A. Kristiadi, and P. Hennig. Fast predictive uncertainty for classification with Bayesian deep networks. *arXiv preprint arXiv:2003.01227*, 2020. 74

[132] S. Hochreiter and J. Schmidhuber. Flat minima. *Neural computation*, 1997. 94

[133] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 1997. 96

[134] E. Hoffer, I. Hubara, and D. Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *NeurIPS*, 2017. 94

[135] M. D. Hoffman and A. Gelman. The No-U-Turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *JMLR*, 2014. 35, 129, 150

[136] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic variational inference. *JMLR*, 2013. 34, 102, 118

[137] Y. Hoffman and E. Ribak. Constrained realizations of Gaussian fields-a simple algorithm. *ApJ*, 1991. 126, 173

[138] A. Honkela and H. Valpola. On-line variational Bayesian learning. In *4th International Symposium on Independent Component Analysis and Blind Signal Separation*, 2003. 76, 93, 102

[139] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *PNAS*, 1982. 20

[140] J. Hron, A. G. d. G. Matthews, and Z. Ghahramani. Variational Gaussian dropout is not Bayesian. *arXiv preprint arXiv:1711.02989*, 2017. 37

[141] J. Hron, A. Matthews, and Z. Ghahramani. Variational Bayesian dropout: pitfalls and fixes. In *ICML*, 2018. 37

[142] Y.-C. Hsu, Y.-C. Liu, A. Ramasamy, and Z. Kira. Re-evaluating continual learning scenarios: A categorization and case for strong baselines. *arXiv preprint arXiv:1810.12488*, 2018. 94

[143] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *CVPR*, 2017. 116

[144] E. Hüllermeier and W. Waegeman. Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods. *Machine Learning*, 2021. 40

[145] M. Humt, J. Lee, and R. Triebel. Bayesian optimization meets Laplace approximation for robotic introspection. *arXiv preprint arXiv:2010.16141*, 2020. 59

[146] F. Huszár. Note on the quadratic penalties in elastic weight consolidation. *PNAS*, 2018. 79, 85, 93

[147] A. Immer, M. Bauer, V. Fortuin, G. Rätsch, and M. E. Khan. Scalable marginal likelihood estimation for model selection in deep learning. In *ICML*, 2021. 74, 96

[148] A. Immer, M. Korzepa, and M. Bauer. Improving predictions of Bayesian neural nets via local linearization. In *AISTATS*, 2021. 74, 96

[149] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015. 21, 23, 71

[150] P. Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov, and A. G. Wilson. Averaging weights leads to wider optima and better generalization. In *UAI*, 2018. 75

[151] P. Izmailov, W. J. Maddox, P. Kirichenko, T. Garipov, D. Vetrov, and A. G. Wilson. Subspace inference for Bayesian deep learning. In *UAI*, 2019. 74, 139

[152] P. Izmailov, S. Vikram, M. D. Hoffman, and A. G. Wilson. What are Bayesian neural network posteriors really like? In *ICML*, 2021. 35, 38, 165

[153] E. Jang, S. Gu, and B. Poole. Categorical reparameterization with gumbel-softmax. In *ICLR*, 2017. 100

[154] S. Jastrzębski, Z. Kenton, D. Arpit, N. Ballas, A. Fischer, Y. Bengio, and A. Storkey. Three factors influencing minima in SGD. *arXiv preprint arXiv:1711.04623*, 2017. 94

[155] M. I. Jordan, Z. Ghahramani, T. S. Jaakkola, and L. K. Saul. An introduction to variational methods for graphical models. *Machine Learning*, 1999. 118

[156] A. G. Journel and C. J. Huijbregts. *Mining geostatistics*. Academic press London, 1978. 126, 173

[157] G. Kahn, A. Villaflor, V. Pong, P. Abbeel, and S. Levine. Uncertainty-aware reinforcement learning for collision avoidance. *arXiv preprint arXiv:1702.01182*, 2017. 37

[158] T.-C. Kao, K. T. Jensen, A. Bernacchia, and G. Hennequin. Natural continual learning: success is a journey, not (just) a destination. *arXiv preprint arXiv:2106.08085*, 2021. 94

[159] T. Karaletsos and T. D. Bui. Hierarchical Gaussian process priors for Bayesian neural network weights. In *NeurIPS*, 2020. 139, 165

[160] T. Karaletsos, P. Dayan, and Z. Ghahramani. Probabilistic meta-representations of neural networks. *arXiv preprint arXiv:1810.00555*, 2018. 162

[161] A. Kendall and Y. Gal. What uncertainties do we need in Bayesian deep learning for computer vision? In *NeurIPS*, 2017. 40, 116

[162] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *ICLR*, 2017. 94

[163] M. E. Khan and H. Rue. Learning algorithms from Bayesian principles. `https://emtiyaz.github.io/papers/learning_from_bayes.pdf`, 2020. Accessed on 14 May 2020. 114

[164] M. E. Khan, D. Nielsen, V. Tangkaratt, W. Lin, Y. Gal, and A. Srivastava. Fast and scalable Bayesian deep learning by weight-perturbation in Adam. In *ICML*, 2018. 34, 73

[165] M. E. Khan, A. Immer, E. Abedi, and M. Korzepa. Approximate inference turns deep networks into Gaussian processes. In *NeurIPS*, 2019. 94, 139

[166] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 21, 27, 92, 109, 130, 154

[167] D. P. Kingma and M. Welling. Auto-encoding variational Bayes. In *ICLR*, 2014. 34, 66, 101, 102, 118

[168] D. P. Kingma, T. Salimans, and M. Welling. Variational dropout and the local reparameterization trick. In *NeurIPS*, 2015. 34, 73, 103, 115, 145, 152

[169] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017. 14

[170] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell. Overcoming catastrophic forgetting in neural networks. *PNAS*, 2017. 15, 54, 76, 80, 84, 85, 87, 93, 95, 178

[171] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell. Reply to huszár: The elastic weight consolidation penalty is empirically valid. *PNAS*, 2018. 93

[172] F. H. Knight. *Risk, uncertainty and profit*, volume 31. Houghton Mifflin, 1921. 40

[173] J. Knoblauch, H. Husain, and T. Diethe. Optimal continual learning has perfect memory and is NP-hard. In *ICML*, 2020. 94

[174] P. W. Koh, S. Sagawa, H. Marklund, S. M. Xie, M. Zhang, A. Balsubramani, W. Hu, M. Yasunaga, R. L. Phillips, I. Gao, et al. Wilds: A benchmark of in-the-wild distribution shifts. In *ICML*, 2021. 39

[175] A. Koroko, A. Anciaux-Sedastrian, I. Gharbia, V. Garès, M. Haddou, and Q. H. Tran. Efficient approximations of the Fisher matrix in neural networks using Kronecker product singular value decomposition. *arXiv preprint arXiv:2201.10285*, 2022. 74

[176] A. Kristiadi, M. Hein, and P. Hennig. Being Bayesian, even just a bit, fixes overconfidence in relu networks. In *ICML*, 2020. 40, 74, 164

[177] A. Kristiadi, M. Hein, and P. Hennig. Learnable uncertainty under Laplace approximations. In *UAI*, 2021. 75

[178] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. 71, 90, 107

[179] A. Krizhevsky, V. Nair, and G. Hinton. CIFAR-10 and CIFAR-100 datasets. https://www.cs.toronto.edu/kriz/cifar.html, 2009. 131

[180] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012. 14, 21, 23

[181] D. Krueger, C.-W. Huang, R. Islam, R. Turner, A. Lacoste, and A. Courville. Bayesian hypernetworks. *arXiv preprint arXiv:1710.04759*, 2017. 139

[182] A. Kucukelbir, D. Tran, R. Ranganath, A. Gelman, and D. M. Blei. Automatic differentiation variational inference. *JMLR*, 2017. 34

[183] F. Kunstner, L. Balles, and P. Hennig. Limitations of the empirical Fisher approximation for natural gradient descent. In *NeurIPS*, 2019. 51

[184] J. Kunze, L. Kirsch, H. Ritter, and D. Barber. Gaussian mean field regularizes by limiting learned information. *Entropy*, 2019.

[185] B. Lakshminarayanan, A. Pritzel, and C. Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *NeurIPS*, 2017. 36, 55, 109, 112, 116, 125, 128, 129

[186] G. Lamb and B. Paige. Bayesian graph neural networks for molecular property prediction. *arXiv preprint arXiv:2012.02089*, 2020. 156

[187] P. S. Laplace. Mémoire sur la probabilité des causes par les évènemens. *Mémoires de Mathematique et de Physique*, 1774. 16, 31

[188] N. Lawrence. *Variational Inference in Probabilistic Models*. PhD thesis, University of Cambridge, 2000. 64

[189] L. LeCam. On some asymptotic properties of maximum likelihood estimates and related Bayes estimates. *University of California Publications in Statistics*, 1953. 32

[190] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1989. 23

[191] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In *NeurIPS*, 1990. 23

[192] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *NeurIPS*, 1990. 54

[193] Y. LeCun, Y. Bengio, et al. Convolutional networks for images, speech, and time series. *The Handbook of Brain Theory and Neural Networks*, 1995. 22

[194] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998. 14, 46, 86, 90, 107

[195] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 2015. 14

[196] J. Lee, J. Sohl-Dickstein, J. Pennington, R. Novak, S. Schoenholz, and Y. Bahri. Deep neural networks as Gaussian processes. In *ICLR*, 2018. 116, 139

[197] J. Lee, H. G. Hong, D. Joo, and J. Kim. Continual learning with extended Kronecker-factored approximate curvature. In *CVPR*, 2020. 94

[198] J. Lee, M. Humt, J. Feng, and R. Triebel. Estimating model uncertainty of neural networks in sparse information form. In *ICML*, 2020. 74, 95

[199] S.-W. Lee, J.-H. Kim, J. Jun, J.-W. Ha, and B.-T. Zhang. Overcoming catastrophic forgetting by incremental moment matching. In *NeurIPS*, 2017. 89, 90, 93

[200] C. Leibig, V. Allken, M. S. Ayhan, P. Berens, and S. Wahl. Leveraging uncertainty information from deep neural networks for disease detection. *Scientific reports*, 2017. 37

[201] T. Lesort, A. Stoian, and D. Filliat. Regularization shortcomings for continual learning. *arXiv preprint arXiv:1912.03049*, 2019. 94

[202] C. Li, C. Chen, D. Carlson, and L. Carin. Preconditioned stochastic gradient Langevin dynamics for deep neural networks. In *AAAI*, 2016. 36

[203] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein. Visualizing the loss landscape of neural nets. In *NeurIPS*, 2018. 55

[204] H. Li, A. Krishnan, J. Wu, S. Kolouri, P. K. Pilly, and V. Braverman. Lifelong learning with sketched structural regularization. *arXiv preprint arXiv:2104.08604*, 2021. 94

[205] Y. Li and Y. Gal. Dropout inference in Bayesian neural networks with alpha-divergences. In *ICML*, 2017. 40, 67, 69

[206] S. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, 1970. 20

[207] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez. A survey on deep learning in medical image analysis. *Medical Image Analysis*, 2017. 15

[208] J. Liu, Z. Lin, S. Padhy, D. Tran, T. Bedrax Weiss, and B. Lakshminarayanan. Simple and principled uncertainty estimation with deterministic deep learning via distance awareness. In *NeurIPS*, 2020. 74

[209] L. Liu and F. Zheng. A Bayesian federated learning framework with multivariate Gaussian product. *arXiv preprint arXiv:2102.01936*, 2021. 94

[210] X. Liu, M. Masana, L. Herranz, J. Van de Weijer, A. M. Lopez, and A. D. Bagdanov. Rotate your networks: Better weight consolidation and less catastrophic forgetting. In *ICPR*, 2018. 94

[211] X. Liu, Y. Li, C. Wu, and C.-J. Hsieh. Adv-bnn: Improved adversarial defense through robust Bayesian neural network. In *ICLR*, 2019. 40

[212] N. Loo, S. Swaroop, and R. E. Turner. Generalized variational continual learning. In *ICLR*, 2021. 93

[213] D. Lopez-Paz and M. Ranzato. Gradient episodic memory for continual learning. In *NeurIPS*, 2017. 95

[214] C. Louizos and M. Welling. Structured and efficient variational deep learning with matrix Gaussian posteriors. In *ICML*, 2016. 73, 162

[215] C. Louizos and M. Welling. Multiplicative normalizing flows for variational Bayesian neural networks. In *ICML*, 2017. 64, 116, 128, 131, 138, 154, 162

[216] C. Louizos, K. Ullrich, and M. Welling. Bayesian compression for deep learning. In *NeurIPS*, 2017. 139

[217] Y. Luo, Z. Huang, Z. Zhang, Z. Wang, M. Baktashmotlagh, and Y. Yang. Learning from the past: Continual meta-learning with Bayesian graph neural networks. In *AAAI*, 2020. 156

[218] C. Ma, Y. Li, and J. M. Hernández-Lobato. Variational implicit processes. In *ICML*, 2019. 139

[219] Y.-A. Ma, T. Chen, and E. B. Fox. A complete recipe for stochastic gradient MCMC. In *NeurIPS*, 2015. 36

[220] D. J. C. MacKay. Bayesian interpolation. *Neural Computation*, 1992. 17, 42

[221] D. J. C. MacKay. A practical Bayesian framework for backpropagation networks. *Neural Computation*, 1992. 73, 74, 96, 145

[222] D. J. C. MacKay. Bayesian neural networks and density networks. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 1995. 42, 116

[223] D. J. C. MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003. 29, 94

[224] C. J. Maddison, A. Mnih, and Y. W. Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *ICLR*, 2017. 100

[225] W. J. Maddox, P. Izmailov, T. Garipov, D. P. Vetrov, and A. G. Wilson. A simple baseline for Bayesian uncertainty in deep learning. In *NeurIPS*, 2019. 75

[226] A. Malinin, N. Band, G. Chesnokov, Y. Gal, M. J. Gales, A. Noskov, A. Ploskonosov, L. Prokhorenkova, I. Provilkov, V. Raina, et al. Shifts: A dataset of real distributional shift across multiple large-scale tasks. In *NeurIPS Datasets and Benchmarks Track*, 2021. 39

[227] S. Mandt, M. D. Hoffman, and D. M. Blei. Stochastic gradient descent as approximate Bayesian inference. *JMLR*, 2017. 75

[228] J. Martens. New insights and perspectives on the natural gradient method. *JMLR*, 2020. 51

[229] J. Martens and R. Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In *ICML*, 2015. 44, 51, 60, 76, 84, 168

[230] A. G. d. G. Matthews, J. Hron, M. Rowland, R. E. Turner, and Z. Ghahramani. Gaussian process behaviour in wide deep neural networks. In *ICLR*, 2018. 116, 139

[231] P. Maybeck. *Stochastic Models, Estimation and Control*, chapter 12.7. Academic press, 1982. 77

[232] M. McCloskey and N. J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation - Advances in Research and Theory*, 1989. 15, 76, 98

[233] D. J. McInerney, L. Kong, K. Arumae, B. Wallace, and P. Bhatia. Kronecker factorization for preventing catastrophic forgetting in large-scale medical entity linking. In *NeurIPS Workshop on Machine Learning in Public Health*, 2021. 94

[234] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, 2017. 16

[235] X. Meng, R. Bachmann, and M. E. Khan. Training binary neural networks using the Bayesian learning rule. In *ICML*, 2020. 114

[236] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 1953. 35

[237] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020. 157

[238] T. P. Minka. Expectation propagation for approximate Bayesian inference. In *UAI*, 2001. 93, 114

[239] M. Minsky and S. Papert. An introduction to computational geometry. *Cambridge tiass., HIT*, 1969. 20

[240] M. Minsky and S. A. Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017. 20

[241] S. I. Mirzadeh, M. Farajtabar, R. Pascanu, and H. Ghasemzadeh. Understanding the role of training regimes in continual learning. In *NeurIPS*, 2020. 94

[242] A. Mishkin, F. Kunstner, D. Nielsen, M. Schmidt, and M. E. Khan. SLANG: Fast structured covariance approximations for Bayesian deep learning with natural gradient. In *NeurIPS*, 2018. 74, 138

[243] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015. 14

[244] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012. 29

[245] D. K. Naik and R. J. Mammone. Meta-neural networks that learn by learning. In *IJCNN*, 1992. 73

[246] T. Nair, D. Precup, D. L. Arnold, and T. Arbel. Exploring uncertainty measures in deep networks for multiple sclerosis lesion detection and segmentation. *Medical image analysis*, 2020. 37

[247] V. Nair and G. E. Hinton. Rectified linear units improve restricted Boltzmann machines. In *ICML*, 2010. 21

[248] E. Nalisnick, J. M. Hernández-Lobato, and P. Smyth. Dropout as a structured shrinkage prior. In *ICML*, 2019. 37

[249] R. M. Neal. Bayesian learning via stochastic dynamics. In *NeurIPS*, 1993. 35, 62

[250] R. M. Neal. *Bayesian Learning for Neural Networks*. PhD thesis, University of Toronto, 1995. 17, 33, 35, 116, 139, 149

[251] R. M. Neal. *Bayesian Learning for Neural Networks*. Springer Science & Business Media, 2012. 35, 108, 145, 150

[252] Y. Nesterov. A Method of Solving a Convex Programming Problem with Convergence Rate O (1/k2). *Soviet Mathematics Doklady*, 1983. 27, 92

[253] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NeurIPS workshop on deep learning and unsupervised feature learning*, 2011. 90, 112

[254] C. V. Nguyen, Y. Li, T. D. Bui, and R. E. Turner. Variational continual learning. In *ICLR*, 2018. 93, 102, 147, 160

[255] H. Nickisch and C. E. Rasmussen. Approximations for binary Gaussian process classification. *JMLR*, 2008. 32

[256] G. K. Nilsen, A. Z. Munthe-Kaas, H. J. Skaug, and M. Brun. Epistemic uncertainty quantification in deep learning classification by the Delta method. *arXiv preprint arXiv:1912.00832*, 2019. 75

[257] J. Nixon, M. W. Dusenberry, L. Zhang, G. Jerfel, and D. Tran. Measuring calibration in deep learning. In *CVPR Workshops*, 2019. 38

[258] S. W. Ober and L. Aitchison. Global inducing point variational posteriors for Bayesian neural networks and deep Gaussian processes. In *ICML*, 2020. 139

[259] M. Opper and O. Winther. A Bayesian approach to on-line learning. *On-line Learning in Neural Networks*, 1998. 76, 77, 93, 102

[260] K. Osawa, S. Swaroop, M. E. Khan, A. Jain, R. Eschenhagen, R. E. Turner, and R. Yokota. Practical deep learning with Bayesian principles. In *NeurIPS*, 2019. 34, 73, 98, 102

[261] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, R. Yokota, and S. Matsuoka. Large-scale distributed second-order optimization using Kronecker-factored approximate curvature for deep convolutional neural networks. In *CVPR*, 2019. 51

[262] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, C.-S. Foo, and R. Yokota. Scalable and practical natural gradient for large-scale deep learning. *TPAMI*, 2020. 51

[263] I. Osband. Risk versus uncertainty in deep learning: Bayes, bootstrap and the dangers of dropout. In *NeurIPS Workshop on Bayesian Deep Learning*, 2016. 36, 37

[264] P. Pan, S. Swaroop, A. Immer, R. Eschenhagen, R. E. Turner, and M. E. Khan. Continual deep learning by functional regularisation of memorable past. In *NeurIPS*, 2020. 94

[265] V. Papyan. The full spectrum of deepnet Hessians at scale: Dynamics with SGD training and sample size. *arXiv preprint arXiv:1811.07062*, 2018. 55

[266] V. Papyan. Traces of class/cross-class structure pervade deep learning spectra. *JMLR*, 2020. 55

[267] H. Park, S.-I. Amari, and K. Fukumizu. Adaptive natural gradient learning algorithms for various stochastic models. *Neural Networks*, 2000. 50

[268] D. B. Parker. Learning-logic: Casting the cortex of the human brain in silicon. Technical report, MIT, 1985. 20

[269] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019. 19, 21, 50, 100, 109, 137, 145

[270] B. A. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 1994. 55

[271] J. W. Peters and M. Welling. Probabilistic binary neural networks. *arXiv preprint arXiv:1809.03368*, 2018. 98, 103, 104

[272] G. Plotkin and M. Pretnar. Handlers of algebraic effects. In *ESOP*, 2009. 152

[273] B. T. Polyak. Some Methods of Speeding up the Convergence of Iteration Methods. *UCMP*, 1964. 27, 92

[274] D. Povey, X. Zhang, and S. Khudanpur. Parallel training of dnns with natural gradient and parameter averaging. *arXiv preprint arXiv:1410.7455*, 2014. 51

[275] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *NeurIPS*, 2007. 74

[276] A. Rahimi and B. Recht. Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning. In *NeurIPS*, 2008. 74

[277] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020. 44

[278] R. Ranganath, S. Gerrish, and D. Blei. Black box variational inference. In *AISTATS*, 2014. 34, 151

[279] R. Ranganath, D. Tran, and D. Blei. Hierarchical variational models. In *ICML*, 2016. 118

[280] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD*, 2020. 44

[281] C. E. Rasmussen and C. K. I. Williams. *Gaussian processes for machine learning*. MIT Press, 2006. 94

[282] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016. 98

[283] R. Ratcliff. Connectionist models of recognition memory: Constraints imposed by learning and forgetting functions. *Psychological Review*, 1990. 76

[284] N. Ravi, J. Reizenstein, D. Novotny, T. Gordon, W.-Y. Lo, J. Johnson, and G. Gkioxari. Accelerating 3d deep learning with PyTorch3D. *arXiv:2007.08501*, 2020. 146

[285] A. Rawat, M. Wistuba, and M.-I. Nicolae. Adversarial phenomenon in the eyes of Bayesian deep learning. *arXiv preprint arXiv:1711.08244*, 2017. 40

[286] S. J. Reddi, S. Kale, and S. Kumar. On the convergence of Adam and beyond. In *ICLR*, 2018. 27

[287] H. Ritter and T. Karaletsos. TyXe: Pyro-based Bayesian neural nets for PyTorch. *Proceedings of Machine Learning and Systems*, 2022. 19

[288] H. Ritter, A. Botev, and D. Barber. Online structured Laplace approximations for overcoming catastrophic forgetting. In *NeurIPS*, 2018. 43

[289] H. Ritter, A. Botev, and D. Barber. A scalable Laplace approximation for neural networks. In *ICLR*, 2018. 43, 76, 93, 138

[290] H. Ritter, M. Kukla, C. Zhang, and Y. Li. Sparse uncertainty representation in deep learning with inducing weights. In *NeurIPS*, 2021. 117

[291] H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 1951. 26

[292] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 1958. 20

[293] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan books, 1962. 20

[294] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, University of California San Diego, 1985. 20, 25

[295] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 1986. 20, 25

[296] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu,

R. Pascanu, and R. Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016. 95

[297] L. Sagun, L. Bottou, and Y. LeCun. Singularity of the Hessian in deep learning. *arXiv preprint arXiv:1611.07476*, 2016. 54

[298] L. Sagun, U. Evci, V. U. Guney, Y. Dauphin, and L. Bottou. Empirical analysis of the hessian of over-parametrized neural networks. *arXiv preprint arXiv:1706.04454*, 2017. 54, 80

[299] T. Salimans, D. Kingma, and M. Welling. Markov chain Monte Carlo and variational inference: Bridging the gap. In *ICML*, 2015. 118

[300] H. Salimbeni and M. Deisenroth. Doubly stochastic variational inference for deep Gaussian processes. In *NeurIPS*, 2017. 139

[301] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry. How does batch normalization help optimization? In *NeurIPS*, 2018. 23

[302] L. J. Savage. *The Foundations of Statistics*. Courier Corporation, 1972. 16

[303] J. Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987. 73

[304] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 2015. 14

[305] J. Schwarz, W. Czarnecki, J. Luketina, A. Grabska-Barwinska, Y. W. Teh, R. Pascanu, and R. Hadsell. Progress & compress: A scalable framework for continual learning. In *ICML*, 2018. 93

[306] A. W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. W. Nelson, A. Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 2020. 14

[307] J. Serrà, D. Surís, M. Miron, and A. Karatzoglou. Overcoming catastrophic forgetting with hard attention to the task. In *ICML*, 2018. 95

[308] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 1948. 41

[309] O. Shayer, D. Levi, and E. Fetaya. Learning discrete weights using the local reparameterization trick. In *ICLR*, 2018. 98, 100, 103, 104, 107, 108, 115

[310] H. Shin, J. K. Lee, J. Kim, and J. Kim. Continual learning with deep generative replay. In *NeurIPS*, 2017. 95

[311] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. 44

[312] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 2016. 14

[313] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 2017. 14

[314] E. Snelson and Z. Ghahramani. Sparse Gaussian processes using pseudo-inputs. In *NeurIPS*, 2006. 94, 116, 119

[315] J. Snoek, Y. Ovadia, E. Fertig, B. Lakshminarayanan, S. Nowozin, D. Sculley, J. Dillon, J. Ren, and Z. Nado. Can you trust your model's uncertainty? Evaluating predictive uncertainty under dataset shift. In *NeurIPS*, 2019. 36, 38, 39, 109, 116

[316] D. Soudry, I. Hubara, and R. Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *NeurIPS*, 2014. 114

[317] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *JMLR*, 2014. 21, 67, 109

[318] J. Staines and D. Barber. Variational optimization. *arXiv preprint arXiv:1212.4507*, 2012. 100

[319] J. Su, M. Cvitkovic, and F. Huang. Sampling-free learning of Bayesian quantized neural networks. *arXiv preprint arXiv:1912.02992*, 2019. 114

[320] S. Sun, C. Chen, and L. Carin. Learning structured weight uncertainty in Bayesian neural networks. In *AISTATS*, 2017. 73

[321] S. Sun, G. Zhang, J. Shi, and R. Grosse. Functional variational Bayesian neural networks. In *ICLR*, 2019. 139

[322] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NeurIPS*, 2014. 14

[323] S. Swaroop, C. V. Nguyen, T. D. Bui, and R. E. Turner. Improving and understanding variational continual learning. *arXiv preprint arXiv:1905.02099*, 2019. 160

[324] J. Swiatkowski, K. Roth, B. S. Veeling, L. Tran, J. V. Dillon, S. Mandt, J. Snoek, T. Salimans, R. Jenatton, and S. Nowozin. The k-tied normal distribution: A compact parameterization of Gaussian mean field posteriors in Bayesian neural networks. In *ICML*, 2020. 34, 73, 116, 128, 138

[325] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013. 14, 40, 67

[326] R. Tanno, D. E. Worrall, A. Ghosh, E. Kaden, S. N. Sotiropoulos, A. Criminisi, and D. C. Alexander. Bayesian image quality transfer with cnns: exploring uncertainty in dMRI super-resolution. In *MICCAI*, 2017. 116

[327] M. Teye, H. Azizpour, and K. Smith. Bayesian uncertainty estimation for batch normalized deep networks. In *ICML*, 2018. 37

[328] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, 2016. 21, 43, 50, 62

[329] S. Thrun and L. Pratt. Learning to learn: Introduction and overview. In *Learning to learn*. Springer, 1998. 73

[330] M. E. Tipping and C. M. Bishop. Probabilistic principal component analysis. *JRSS B*, 1999. 142

[331] M. K. Titsias. Variational learning of inducing variables in sparse Gaussian processes. In *AISTATS*, 2009. 94, 116, 119

[332] M. K. Titsias and M. Lázaro-Gredilla. Doubly stochastic variational Bayes for non-conjugate inference. In *ICML*, 2014. 34, 102, 118

[333] M. K. Titsias, J. Schwarz, A. G. d. G. Matthews, R. Pascanu, and Y. W. Teh. Functional regularisation for continual learning with Gaussian processes. In *ICLR*, 2020. 94

[334] M. Tomczak, S. Swaroop, and R. E. Turner. Efficient low rank Gaussian variational inference for neural networks. *NeurIPS*, 2020. 74

[335] M. Tomczak, S. Swaroop, A. Foong, and R. E. Turner. Collapsed variational bounds for Bayesian neural networks. In *NeurIPS*, 2021. 105

[336] D. Tran, M. D. Hoffman, D. Moore, C. Suter, S. Vasudevan, and A. Radul. Simple, distributed, and accelerated probabilistic programming. In *NeurIPS*, 2018. 106

[337] D. Tran, M. D. Hoffman, D. Moore, C. Suter, S. Vasudevan, A. Radul, M. Johnson, and R. A. Saurous. Simple, distributed, and accelerated probabilistic programming. In *NeurIPS*, 2018. 145

[338] D. Tran, M. Dusenberry, M. van der Wilk, and D. Hafner. Bayesian layers: A module for neural network uncertainty. In *NeurIPS*, 2019. 146, 152, 161, 162

[339] B. Trippe and R. E. Turner. Overpruning in variational Bayesian neural networks. *arXiv preprint arXiv:1801.06230*, 2018. 35, 98, 154

[340] N. Tselepidis, J. Kohler, and A. Orvieto. Two-level K-FAC preconditioning for deep learning. *arXiv preprint arXiv:2011.00573*, 2020. 74

[341] H. Tseran, M. E. Khan, T. Harada, and T. D. Bui. Natural variational continual learning. In *NeurIPS Continual Learning Workshop*, 2018. 93

[342] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016. 14, 98

[343] A. W. van der Vaart. *Asymptotic Statistics*. Cambridge university press, 1998. 32

[344] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *NeurIPS*, 2017. 14, 96

[345] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 2019. 14

[346] M. Wainwright and M. Jordan. Graphical models, exponential families, and variational inference. *FTML*, 2008. 33, 101

[347] A. M. Walker. On the asymptotic behaviour of posterior distributions. *JRSS B*, 1969. 32

[348] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019. 146, 148, 155

[349] S. Watanabe. Generalized Bayesian framework for neural networks with singular Fisher information matrices. In *NOLTA*, 1995. 33

[350] S. Watanabe. Almost all learning machines are singular. In *FOCI*, 2007. 33

[351] M. Welling and Y. W. Teh. Bayesian learning via stochastic gradient Langevin dynamics. In *ICML*, 2011. 36

[352] Y. Wen, P. Vicol, J. Ba, D. Tran, and R. Grosse. Flipout: Efficient pseudo-independent weight perturbations on mini-batches. In *ICLR*, 2018. 152

[353] Y. Wen, D. Tran, and J. Ba. BatchEnsemble: An alternative approach to efficient ensemble and lifelong learning. In *ICLR*, 2020. 116, 128, 138

[354] F. Wenzel, K. Roth, B. S. Veeling, J. Światkowski, L. Tran, S. Mandt, J. Snoek, T. Salimans, R. Jenatton, and S. Nowozin. How good is the Bayes posterior in deep neural networks really? In *ICML*, 2020. 58, 98

[355] F. Wenzel, K. Roth, B. S. Veeling, J. Świątkowski, L. Tran, S. Mandt, J. Snoek, T. Salimans, R. Jenatton, and S. Nowozin. How good is the Bayes posterior in deep neural networks really? In *ICML*, 2020. 165

[356] P. J. Werbos. *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. PhD thesis, Harvard University, 1975. 20

[357] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992. 34, 100

[358] A. G. Wilson. The case for Bayesian deep learning. *arXiv preprint arXiv:2001.10995*, 2020. 36

[359] A. G. Wilson and P. Izmailov. Bayesian deep learning and a probabilistic perspective of generalization. In *NeurIPS*, 2020. 36

[360] D. Wingate and T. Weber. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299*, 2013. 151

[361] A. Wu, S. Nowozin, E. Meeds, R. E. Turner, J. M. Hernández-Lobato, and A. L. Gaunt. Deterministic dariational inference for robust Bayesian neural networks. In *ICLR*, 2019. 34, 73, 99

[362] Y. Wu, X. Zhu, C. Wu, A. Wang, and R. Ge. Dissecting Hessian: Understanding common structure of Hessian in neural networks. *arXiv preprint arXiv:2010.04261*, 2020. 55

[363] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017. 90, 112

[364] H. Xu, B. Liu, L. Shu, and P. Yu. BERT: Post-training for review reading comprehension and aspect-based sentiment analysis. In *ACL*, 2019. 116

[365] Z. Yao, A. Gholami, K. Keutzer, and M. W. Mahoney. PyHessian: Neural networks through the lens of the Hessian. In *IEEE BigData*, 2020. 55

[366] P. Yap, H. Ritter, and D. Barber. Addressing catastrophic forgetting in few-shot problems. In *ICML*, 2021.

[367] D. Yin, M. Farajtabar, A. Li, N. Levine, and A. Mott. Optimization and generalization of regularization-based continual learning: a loss approximation viewpoint. *arXiv preprint arXiv:2006.10974*, 2020. 94, 95

[368] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *NeurIPS*, 2014. 16

[369] S. Zagoruyko and N. Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016. 71

[370] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012. 27

[371] F. Zenke, B. Poole, and S. Ganguli. Continual learning through synaptic intelligence. In *ICML*, 2017. 85, 87, 92, 95, 160, 178

[372] C. Zhang, J. Bütepage, H. Kjellström, and S. Mandt. Advances in variational inference. *TPAMI*, 2018. 33, 118

[373] G. Zhang, S. Sun, D. Duvenaud, and R. Grosse. Noisy natural gradient as variational inference. In *ICML*, 2018. 74, 138

[374] R. Zhang, C. Li, J. Zhang, C. Chen, and A. G. Wilson. Cyclical stochastic gradient MCMC for Bayesian deep learning. In *ICLR*, 2020. 36, 116

[375] Y. Zhang, S. Pal, M. Coates, and D. Ustebay. Bayesian graph convolutional neural networks for semi-supervised classification. In *AAAI*, 2019. 156

[376] Z. Zhu, J. Wu, B. Yu, L. Wu, and J. Ma. The anisotropic noise in stochastic gradient descent: Its behavior of escaping from sharp minima and regularization effects. In *ICML*, 2019. 94