M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Bayesian Deep Learning

Ramiro Mata

Kongens Lyngby 2018

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

Quantifying model uncertainty in deep neural networks remains a challenge in the field. Bayesian Neural Networks (BNNs) offer a mathematically principled approach to model uncertainty by treating the model's weights as stochastic variables, and thereby effectively learning a distribution over the different functions the model can represent. Recently there's been much attention in Variational Inference (VI) schemes that achieve approximate Bayesian inference in deep learning. However, these methods require extensive changes to code bases, require at least twice the memory and computation requirements, and can be difficult to implement. Here we explore in detail VI schemes that don't suffer from these limitations, are fast and scalable, and which achieve approximate Bayesian inference through the optimizer. These algorithms resemble RMSPROP and Adam, hence their name Vprop and Noisy Adam. We show empirically through a series of experiments that these models are data-efficient, resilient to overfitting, and are able to learn better from noisy data relative to their deterministic neural network counterparts. Further, we evaluate sparse-inducing hierarchical scale priors such as the gamma, and the Horseshoe on Black-Box Variational Inference (BBVI) models, and discuss the challenges of generalizing these hierarchical priors to Noisy Adam and to natural gradient methods in general. Finally, we apply these BNN models to Alipes Capital's dataset for the task of stock price prediction and show that 1) we are able to marginally beat their deep learning benchmark model, and 2) quantifying the model's uncertainty in this task opens the opportunity to better gauge risk during trade execution and to better understand in which scenarios the model becomes uncertain, such as *out-of-distribution* test samples.

Preface

This Master thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a Master of Science in Engineering (M.Sc.Eng.) degree in Mathematical Modelling and Computation.

[illegible]

Kongens Lyngby, September 27, 2018

Ramiro Mata

Acknowledgements

I would like to thank Carsten Stahlhut for his great support during our stay at Alipes Capital. We spent many hours digging through theory and putting it at work to real world applications - I thank him for his patience, guidance and expertise in the subject matter. I would also like to thank Ole Winther for his academic support and helping us navigate the unexpected obstacles that tend to come in a master thesis project. Finally, I am extremely grateful for Alipes Capital for granting me this fruitful opportunity, and to their staff in general for making me feel welcome, helping me navigate through their software and hardware infrastructure, and sharing with us the amazing meals and desserts Louise Ellegaard, the chef, prepared for everyone to keep us going through the day.

Contents

CHAPTER 1
Introduction

1.1 Introduction

Deep learning has attracted much attention in the artificial intelligence community in recent years by improving state-of-the-art in domains varying from video object detection, speech recognition, text and genomics. Yet, capturing model uncertainty remains a challenge in the field. But, how is this relevant to the real world? While adoption of deep learning in industry is becoming more wide-spread, there are numerous industries, such as in medicine, autonomous driving, finance, etc., where understanding the uncertainty behind a model's predictions is crucial.

A probabilistic approach to machine learning allows for uncertainty to be used as part of the data analysis and decision making process. In some applications, lung cancer diagnosis based on MRI imagery for instance, it is paramount to know how confident the model is about its output; or in autonomous driving for instance, it is necessary to know how certain the model is about its object depth perception when driving in the presence of pedestrians. When the practitioner has knowledge about the model's uncertainty, it allows him/her to a) pinpoint which situations the model is uncertain in, b) assess whether more diverse data is needed to reduce the model's uncertainty, and c) be able to deploy the model only when the uncertainty is below the practitioner's tolerance. Further, model uncertainty is not only beneficial to practitioners, but also to reinforcement learning agents. The exploration-exploitation strategy of an agent is key in maximizing rewards. Previous studies have shown that incorporating uncertainty allows the agent to better learn how to trade-off exploration and exploitation ([Kha18], [Blu15]).

As of now, the most prevalent practice in deep learning is treating the model as a deterministic function with point estimates for both its parameters and its predictions - uncertainty is an afterthought. This lack of robust uncertainty estimates combined with its value and necessity in a wide array of domains is one of the motivating factors for Bayesian Deep Learning. Bayesian Deep Learning offers a mathematically principled approach to quantifying uncertainty in a model's predictions. It does so by treating the weights of a model as stochastic variables, and thereby capturing a distribution over the functions the model can represent. In practice, however, Bayesian Deep Learning faces its own challenges. Doing exact Bayesian inference in deep neural networks is computationally intractable, which prevents theory to be applicable in practice. The high computational cost of doing exact Bayesian inference limits its speed and scalability even for most practical applications. Recently, however, a renewed interest in approximate inference techniques for Bayesian Deep Learning that circumvent these limitations has emerged. These methods fall under Markov Chain Monte Carlo (MCMC) and Variational Inference (VI) schemes. Here we focus on VI methods - for an overview of MCMC methods, see [Rad95], [TW15] and [Hof17]. In this work, we focus on a class of VI algorithms that through techniques such as the reparameterization trick and Price's and Bonnet's theorem, allow us to reduce the storage and computational costs by half making them faster and more scalable. Moreover, they also allow us to reduce the complexity of code implementation, making them very adaptable to prominent deep learning frameworks such as `Pytorch` and `Tensorflow`. We evaluate this class of algorithms on benchmark datasets as well as in simulation experiments. Finally, we elaborate on the added benefits of using natural gradients when optimizing in the space of probability distributions, and explore the challenges of generalizing these natural gradient methods to hierarchical priors.

CHAPTER 2 Variational Inference for Neural Networks

Foundations In this chapter we briefly review the basic framework of Bayesian neural networks, variational inference, mean field theory and natural gradients. We dedicate a section to each. These will serve as foundational blocks for understanding the theory behind the algorithms that take center stage in this thesis: Vprop and Noisy Adam. Further, they will serve as reference points for future sections and analyses.

Let $\mathcal{D} = \{X_i, y_i\}_{i=1}^{N}$ denote a dataset with inputs X_i and targets y_i. We then define a neural network as a probabilistic model $p(y|X, \boldsymbol{w})$ with a set of weights $\boldsymbol{w} = \{w_i\}_{i=1}^{M}$. In a deterministic setting, the neural network is then trained by optimizing the log-likelihood which we define as

$$\log p(\mathcal{D}|\boldsymbol{w}) = \sum_i \log p(y_i|X_i, \boldsymbol{w}) \tag{2.1}$$

which is achieved by changing the weights iteratively through a gradient descent optimization scheme.

In a Bayesian setting, we consider the weights $\boldsymbol{w}$ to be random variables drawn from some unknown posterior distribution. If we associate a prior distribution $p(\boldsymbol{w}|\boldsymbol{\alpha})$, where $\boldsymbol{\alpha}$ denotes the parameters of the distribution, to the weights, we obtain the posterior distribution denoted as $P(\boldsymbol{w}|\mathcal{D}, \boldsymbol{\alpha})$. The goal then becomes computing the true posterior distribution, which typically is intractable. To realize this, note that by using Bayes' theorem we can write (we suppress $\boldsymbol{\alpha}$ from the notation)

$$P(\boldsymbol{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{w})p(\boldsymbol{w})}{p(\mathcal{D})} \tag{2.2}$$

From this expression, the known entities are the prior $p(\boldsymbol{w})$ which we know the exact form of, and the likelihood $p(\mathcal{D}|\boldsymbol{w})$ which can be computed as well. However, if we wanted to compute $p(\mathcal{D})$ which is typically called the evidence, we would compute its marginalization over the weights $\boldsymbol{w}$ as follows:

$$p(\mathcal{D}) = \int_w p(\mathcal{D}|\boldsymbol{w})p(\boldsymbol{w})d\boldsymbol{w} \tag{2.3}$$

In most cases, the integral cannot be computed analytically or requires exponential time to compute, making it intractable. Hence, another approach such as variational inference is required to obtain at least an approximation of the true posterior.

2.1 Variational Inference

Variational Inference (VI) is a way of solving the problem of computing the true posterior in (2.2) by turning the inference problem into an optimization problem. Variational inference does not compute the posterior exactly but instead gives an approximation. Hence VI assumes an approximate posterior distribution defined as $q(\boldsymbol{w}|\boldsymbol{\theta})$ which we call the variational posterior with $\boldsymbol{\theta}$ being the parameters of the variational posterior. Changing the parameters $\boldsymbol{\theta}$ then gives us different approximations to the true posterior. We now seek the approximate posterior distribution which is closest to the true posterior. This can be done by considering the KL divergence given as

$$\mathrm{KL}(q(\boldsymbol{w})||p(\boldsymbol{w}|\mathcal{D})) = \mathbb{E}_q(\log q(\boldsymbol{w})) - \mathbb{E}_q(\log p(\boldsymbol{w}|\mathcal{D})) \tag{2.4}$$

which is a measure of how one distribution diverges from the other and can be roughly interpreted as the distance between two distributions (though this is not entirely correct). However, computing (2.4) would require computing the log evidence since the term

$$\begin{aligned}
\mathbb{E}_q(\log p(\boldsymbol{w}|\mathcal{D})) &= \mathbb{E}_q\left(\log \frac{p(\boldsymbol{w},\mathcal{D})}{p(\mathcal{D})}\right) && (2.5)\\
&= \mathbb{E}_q(\log p(\boldsymbol{w},\mathcal{D})) - \mathbb{E}_q(\log p(\mathcal{D})) && (2.6)\\
&= \mathbb{E}_q(\log p(\boldsymbol{w},\mathcal{D})) - \log p(\mathcal{D}) && (2.7)
\end{aligned}$$

contains the log evidence as seen in (2.7). To get around this, we instead compute the evidence lower bound, in short ELBO, which is equivalent up to a constant and is given as

$$\begin{aligned}
\log p(\mathcal{D}) &\geq \mathbb{E}_q(\log p(\boldsymbol{w},\mathcal{D})) - \mathbb{E}_q(\log q(\boldsymbol{w})) && (2.8)\\
&= \mathbb{E}_q(\log[p(\mathcal{D}|\boldsymbol{w}) \cdot p(\boldsymbol{w})]) - \mathbb{E}_q(\log q(\boldsymbol{w})) && (2.9)\\
&= \mathbb{E}_q(\log p(\mathcal{D}|\boldsymbol{w})) + \mathbb{E}_q(\log p(\boldsymbol{w})) - \mathbb{E}_q(\log q(\boldsymbol{w})) && (2.10)\\
&= \mathbb{E}_q(\log p(\mathcal{D}|\boldsymbol{w})) - \mathrm{KL}(q(\boldsymbol{w})||p(\boldsymbol{w})) && (2.11)
\end{aligned}$$

From (2.11) it becomes clear that the ELBO depends on the likelihood (as in the deterministic case) and a negative KL divergence term between a prior of our choosing

$p(\boldsymbol{w})$ and the variational posterior. Thus, the KL term can be seen as a regularizer term since it will push the posterior towards the prior whilst the likelihood term seeks to optimize the weights in such a way that they best explain the data. Lastly, as seen in (2.8) the ELBO is a lower bound for the log evidence since KL ≥ 0 [KL51]. Hence, we seek to maximize the ELBO since we want as tight a bound as possible on the evidence.

Remark Throughout this thesis we are considering the case of minimizing the ELBO. Thus we consider the negative log-likelihood and a positive KL divergence term, i.e. $\log p(\mathcal{D}) \geq -\mathbb{E}_q(\log p(\mathcal{D}|\boldsymbol{w})) + \mathrm{KL}(q(\boldsymbol{w})||p(\boldsymbol{w}))$.

2.2 Gaussian Mean Field Approximation

In the Bayesian setting we are interested in estimating the full posterior distribution $p(\boldsymbol{w}|\mathcal{D})$ rather than a point estimate under the Maximum Likelihood framework. One of the complications in estimating the full true posterior distribution is the number of parameters that need to be trained. For a Gaussian distribution, the number of parameters to be learned is $n + n^2$, where n is the number of weights in the neural network. The n stems from the $\boldsymbol{\mu}$ vector while the n^2 stems from the full covariance matrix, $\boldsymbol{\Sigma}$. In most practical deep learning applications n can be in the order of millions, and so estimating a full covariance matrix is prohibitive. A mean field approximation allows us to reduce the number of parameters to be learned to $2n$. I.e., using a mean field approximation allows us to trade-off model expressivity for scaling to high dimensions. We do this by using a fully factorized gaussian distribution

$$q(\boldsymbol{w}) = q(w_1, w_2, w_3, \ldots, w_n) = \prod_{i=1}^{n} q(w_i) \tag{2.12}$$

As such, our variational posterior takes the form

$$q(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{\mu}, \mathrm{diag}(\Sigma)) = \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{\mu}, \boldsymbol{\sigma^2}) \tag{2.13}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2 = \mathrm{diag}(\boldsymbol{\Sigma})$ are vectors whose elements correspond to the μ and σ^2 of individual weights, respectively. As such, each weight is independent of the others and is fully characterized by its own parameters. Thus, our mean field approximation is a multivariate Gaussian with diagonal covariance (equation (2.13)). In the work carried out in this thesis, both our prior, $p(\boldsymbol{w})$, and our variational distribution, $q(\boldsymbol{w})$, will for the most part take this form, though we explore some models where this is not the case.

Remark When using a mean field distribution for our variational distribution, we make predictions by using the *maximum a posteriori* (MAP) estimate of each weight distribution, $q(w_i)$, as the final weights of our neural network. The MAP estimate is defined as the mode of the posterior probability distribution. In the Gaussian case, the mode is equivalent to its mean. Thus, when making predictions, we use the mean as our MAP estimate.

2.3 Natural Gradient: The Steepest Descent Direction in Riemannian Manifolds

As seen in the previous sections, at its core, variational inference optimizes in the space of probability distributions. The optimization becomes a balance between finding a set of weights that explain our data through the log-likelihood, but which also stay as close to our prior by pushing the $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ of our variational posterior towards it through the Kullback-Liebler divergence term in (2.11). As such, we must ask: *"Is ordinary gradient descent appropriate for optimizing in the space of probability distributions?"*

In ordinary gradient descent, the gradient captures the direction where a change in the network's weights ($\boldsymbol{w}$) leads to the largest change in the output of a loss function, here denoted $\boldsymbol{L}(\boldsymbol{w})$ (where both changes in weights and function space are measured in Euclidean space). We formalize this below.

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha \nabla_{\boldsymbol{w}} \boldsymbol{L}(\boldsymbol{w}) \tag{2.14}$$

However, in some cases the weight space has a certain underlying structure such that the gradient does not point in the direction of steepest descent. One of these cases is in Bayesian neural networks. Since it is parameters of a probability distribution which are being optimized, the solution space is not Euclidean and has a different Riemannian metric structure. In such cases, using the ordinary gradient to optimize the parameters of probability distributions may not be optimal. Figure 2.1 illustrates this point. It shows that while the Euclidean distance between the two Gaussians in the upper and lower plot is the same, their similarity is evidently different.

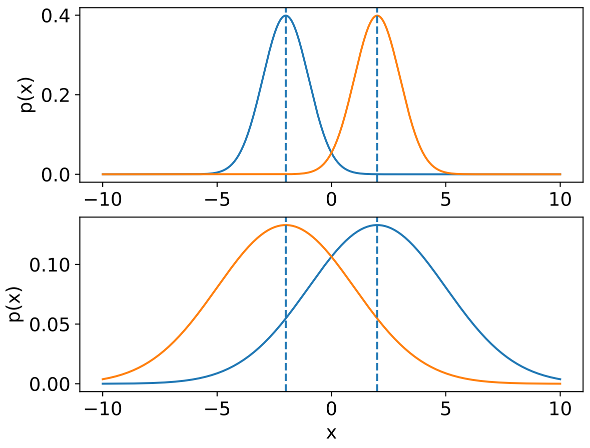


Figure 2.1: *The upper subplot shows two Gaussian distributions centered at* $\mu_1 = -2$ *and* $\mu_2 = 2$ *both with equal variances,* $\sigma_1^2 = \sigma_2^2 = 1$ *(Euclidean distance is 4). The bottom subplot shows exactly the same two Gaussians as above but where the variances have increased by 8,* $\sigma_1^2 = \sigma_2^2 = 9$*. In both cases, the Euclidean distance in parameter space is 4 (*$|\mu_1 - \mu_2|$*). However, it is evident that the distributions in the bottom plot overlap more and thus are more similar to each other. This figure highlights the inappropriateness of using Euclidean distance in the space of probability distributions.*

Indeed, this is a motivation for using natural gradient descent. Formally the natural gradient is defined as

$$\widetilde{\nabla}\boldsymbol{L}(\boldsymbol{w}) = \boldsymbol{G}^{-1}\nabla\boldsymbol{L}(\boldsymbol{w}) \tag{2.15}$$

where $\boldsymbol{G}$ is the Riemannian metric tensor associated with the given manifold, e.g. in Euclidean space it is the standard Euclidean metric (given in (2.17)). Thus, when the loss function has a structure best represented by a curved manifold, the natural gradient is the steepest direction in that Riemannian manifold. See figure 2.2 for a geometric illustration of the gradient projection from the manifold onto Euclidean space. To elaborate on this, consider the following derivation from [Ama98]. Let $S = \{\boldsymbol{w} \in \mathbb{R}^n\}$ be the solution space where the loss function, $\boldsymbol{L}(\boldsymbol{w})$, is defined, and where $\boldsymbol{w}$ are the weights of our neural network being optimized. The squared length of the incremental vector $d\boldsymbol{w}$ connecting $\boldsymbol{w}$ and $\boldsymbol{w} + d\boldsymbol{w}$ in space S can be formulated as:

$$d\boldsymbol{w}^2 = \sum_{i,j} g_{ij}(\boldsymbol{w}) d\boldsymbol{w}_i d\boldsymbol{w}_j \tag{2.16}$$

where $g_{ij}(\boldsymbol{w})$ is an indicator function as defined in (2.17). When space S is Euclidean and has an orthonormal coordinate system, the Riemannian metric tensor $\boldsymbol{G} = (g_{ij})$

is simply the identity matrix with

$$g_{ij}(\boldsymbol{w}) = \left\{ \begin{array}{cc} 1, & i = j \\ 0 & i \neq j \end{array} \right\} \tag{2.17}$$

In this case, we can see that the natural gradient is equivalent to the ordinary gradient:

$$\begin{aligned} \widetilde{\nabla}\boldsymbol{L}(\boldsymbol{w}) &= \boldsymbol{G}^{-1}\nabla\boldsymbol{L}(\boldsymbol{w}) && (2.18) \\ &= \boldsymbol{I}\nabla\boldsymbol{L}(\boldsymbol{w}) && (2.19) \\ &= \boldsymbol{L}(\boldsymbol{w}) && (2.20) \end{aligned}$$

However, when the space S is the parameter space of probability distributions, the Riemannian metric is best defined by the Fisher Information [Ama98]. The Fisher Information is given by

$$\begin{aligned} \boldsymbol{F_w} &= \mathbb{E}_p[\nabla_{\boldsymbol{w}} \log p(\mathcal{D}|\boldsymbol{w})(\nabla_{\boldsymbol{w}} \log p(\mathcal{D}|\boldsymbol{w}))^T] && (2.21) \\ &= \mathrm{Cov}[\nabla_{\boldsymbol{w}} \log p(\mathcal{D}|\boldsymbol{w})] && (2.22) \\ &= \mathbb{E}_p[-(\nabla_{\boldsymbol{w}} \log p(\mathcal{D}|\boldsymbol{w}))^2] && (2.23) \end{aligned}$$

In this regard, the Fisher Information Matrix (FIM) can be seen as an approximation to the Hessian of the log likelihood ($\log p(\mathcal{D}|\boldsymbol{w})$). However, unlike the Hessian, which is not necessarily always positive semi-definite in non-convex problems, the FIM is always positive semi-definite. Using the FIM as the Riemannian metric in (2.18) and scaling the ordinary gradient with the inverse of the FIM gives rise to the natural gradient in the space of statistical models:

$$\widetilde{\nabla}\boldsymbol{L}(\boldsymbol{w}) = \boldsymbol{F}^{-1}\nabla_{\boldsymbol{w}}\boldsymbol{L}(\boldsymbol{w}) \tag{2.24}$$

where the natural gradient update then becomes (where α is a step size/learning rate):

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha\boldsymbol{F}^{-1}\nabla_{\boldsymbol{w}}\boldsymbol{L}(\boldsymbol{w}) \tag{2.25}$$

Natural Gradient as a Gauss-Newton Saddle Free Method Notice that this update rule resembles a Newton update but where in this case the gradient is scaled by the inverse of the FIM rather than the Hessian. Indeed, this natural gradient method can be regarded as a Gauss-Newton method [Ama16]. Interestingly, recent studies have shown that most critical points in high-dimensional landscapes are saddle points rather than local minima [Dau+14]. When optimizing neural networks, for instance, Dauphin et al. argue that the optimization landscape is proliferated with saddle points surrounded by high error plateaus that can significantly slow down learning. Sometimes these can give the impression of convergence to a local minimum. While the Newton method is able to escape rapidly from plateaus via curvature information from the Hessian, it is prone to getting stuck in saddle points. That is, saddle points become attractors under Newtonian dynamics. Dauphin et al. propose a method that takes the absolute values of the Hessian's eigenvalues which allows it to behave as the Newton method, but with the added advantage that it becomes repellent to saddle points - they term their algorithm Saddle Free Newton (SFN). Interestingly, Amari [Ama16] recasts the SFN method as a natural gradient method, and shows that whether using an absolute eigenvalued Hessian or the FIM as the Riemannian metric, these natural gradient methods are able to overcome saddle points as the scaling matrices are positive semi-definite by construction.

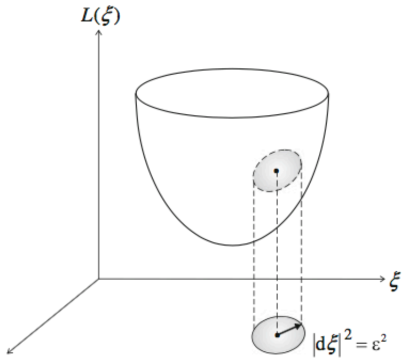


Figure 2.2: *Figure taken from [Ama16] shows a geometric view of the projection of the natural gradient in the Riemannian Manifold (round cone) onto Euclidean space.*

2.4 Achieving Variational Inference in Bayesian Neural Networks

Various variational inference methods have been developed and used for Bayesian neural networks. The most prominent methods are characterized by using either the REINFORCE method, also known as the log-derivative trick, or the reparameterization trick method to estimate gradients of the ELBO.
For a classic feedforward Bayesian neural network, for instance, methods such as Black Box Variational Inference (BBVI) and Bayes By Backprop (BBB) [Blu15] proposed by Blundell et al. perform variational inference. However, these Bayesian methods suffer from two main draw-backs: 1) they require at least twice the amount of gradient computations, and 2) implementing them in available deep learning libraries is difficult as it requires reconstructing the basic units of the model: the layers and weights of a neural network.

If we consider e.g. Bayes By Backprop, the idea behind it is to assume that all the weights $\boldsymbol{w}$ of the network come from a distribution. The variational posterior of the weights then becomes $q(\boldsymbol{w}|\boldsymbol{\theta})$ following the notation from section 2.1. Now, by employing the reparameterization trick we can write an arbitrary weight as $w = \mu + \sigma \cdot \varepsilon$ where μ and σ are the parameters of the variational posterior for that weight. The reparameterization trick enables us to make the parameters of the variational posterior trainable. Thus, we can learn the optimal variational posterior distribution by iteratively sampling weights through the reparameterization trick, compute the loss, in our case the ELBO, take gradients of this with respect to the parameters of the variational posterior and use these to update the parameters. This, in essence, is Bayes by Backprop, and has been shown to work quite well, making Bayesian neural networks resistant to overfitting.

However, Bayes by Backprop does have the drawbacks previously mentioned. Typically, implementing Bayes by Backprop requires significant changes to existing code bases, be it `Tensorflow`, `PyTorch` or other. Take e.g. a feed-forward neural network. The implementation of Bayes by Backprop requires the deterministic weights at each layer to be changed to random variables, utilizing the reparameterization trick. This means that the programmer has to change the way layers are constructed. Furthermore, computing the ELBO requires additional changes to the code base.

As examplified in [Kha17], another drawback is the increasing number of parameters which need to be optimized. Since weights are obtained through the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$, the number of parameters to be optimized is doubled in the case of a mean field approximation and squared in the case of a full covariance matrix. This is due to the fact that we are optimizing the parameters of the entire distribution and not just point estimates of the weights. Lastly, issues such as $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ being two different quantities can potentially require different types of tuning. For instance, the gradient with respect to $\boldsymbol{\sigma}$ might demand a different implementation to avoid numerical issues. Indeed, BBVI methods suffer from these drawbacks in general. In the next section, we review in detail a class of algorithms which circumvent many of the limitations inherent to BBVI, Bayes by Backprop and other methods in Bayesian Deep Learning.

2.5 Variational Inference via Optimizers

New ways of obtaining variational inference for Bayesian neural networks that address the issues mentioned in the previous section have recently appeared. By a clever use of Price's and Bonnet's theorem and making connections between VI updates and existing popular gradient-descent methods such as Adam and RMSprop, these methods are able to circumvent many of the limitations of previous methods in Bayesian Deep Learning. Specifically, they are able to use current deep learning frameworks, such as `Pytorch`, `Tensorflow`, etc., without making any modifications by simply moving the variational inference scheme inside the optimizer. These methods remove the need for creating custom layers in a network as well as the need for explicitly changing the loss to the ELBO. The two papers proposing these new methods, which will be the main focus of this thesis are Vprop: Variational Inference using RMSprop by Mohammad Emtiyaz Khan et al. and Noisy Natural Gradient as Variational Inference by Guodong Zhang et al.

2.5.1 Noisy Natural Gradient as Variational Inference

Zhang et al. present two optimizers which perform variational inference. Namely, noisy Adam and noisy KFAC, which resemble Adam and KFAC to some extent. The algorithms are presented in tables 2.1 and 2.2 (see appendix A for the original optimizers). In this section we will review and highlight key points of the two optimizers and how they have been derived.

2.5.1.1 Noisy Adam

In their paper, they derive the noisy Adam algorithm through a connection between natural gradient for point estimation (NGPE) and natural gradient for variational inference (NGVI). NGPE acts in a deterministic setting, where we seek to optimize some cost function f, such as the likelihood of the data. NGPE is defined as:

$$\widetilde{\nabla}_{\boldsymbol{w}} f = \boldsymbol{F}^{-1} \nabla_{\boldsymbol{w}} f \tag{2.26}$$

with $\boldsymbol{F}$ being the Fisher matrix, defined as $\boldsymbol{F} = \mathrm{Cov}_{\boldsymbol{x} \sim p_{\mathcal{D}}, y \sim p(y|\boldsymbol{x},\boldsymbol{w})}[\nabla_{\boldsymbol{w}} \log p(y|\boldsymbol{x}, \boldsymbol{w})]$. That is, the covariance of the score function between $\boldsymbol{x}$, sampled from the data distribution and y, sampled from the predictive distribution. Likewise, NGVI is defined as

$$\widetilde{\nabla}_{\boldsymbol{\phi}} \mathcal{L} = \boldsymbol{F}^{-1} \nabla_{\boldsymbol{\phi}} \mathcal{L} \tag{2.27}$$

where the gradient is now of the ELBO, with respect to the parameters of the variational posterior and $\boldsymbol{F}$ is now the Fisher matrix of the variational posterior q, i.e. $\boldsymbol{F} = \mathrm{Cov}_{\boldsymbol{w} \sim q}[\nabla_{\boldsymbol{\phi}} \log q(\boldsymbol{w}; \boldsymbol{\phi})]$.

As the paper shows, it turns out that the connection between NGPE and NGVI is that one can approximate NGVI updates with a variant of NGPE updates with adaptive weight noise, which results in a method which they term Noisy Natural Gradient (NNG). They derive NNG by firstly obtaining NGVI updates through a trust region optimization problem of finding the optimal parameter θ in a neighborhood of θ_0 defined with KL divergence [Zha17]. Specifically, for a model parameterized by $\boldsymbol{\theta}$ on a subspace $\mathcal{S}$ we have the optimization problem

$$\arg\min_{\theta \in \mathcal{S}} \alpha (\nabla_{\theta} h)^T \boldsymbol{\theta} + \mathrm{KL}(p_{\boldsymbol{\theta}} || p_{\boldsymbol{\theta}_0}) \tag{2.28}$$

for some differentiable function $h(\boldsymbol{\theta})$. The optimal solution is given as $\boldsymbol{\theta} - \alpha \boldsymbol{F}^{-1} \nabla_\theta h$ where $\boldsymbol{F}$ is the Fisher matrix. They use this approach to compute the natural gradient for a multivariate Gaussian by computing $\mathrm{KL}(\mathcal{N}||\mathcal{N}_0)$. Thus they obtain the Fisher matrix for the mean and covariance of a Gaussian given as

$$\boldsymbol{F}_\mu = \nabla^2_\mu \mathrm{KL}(\mathcal{N}||\mathcal{N}_0) = \boldsymbol{\Sigma}_0^{-1} \approx \boldsymbol{\Sigma}^{-1} \tag{2.29}$$

$$\boldsymbol{F}_\Sigma = \nabla^2_\Sigma \mathrm{KL}(\mathcal{N}||\mathcal{N}_0) = \frac{1}{2} \boldsymbol{\Sigma}^{-1} \otimes \boldsymbol{\Sigma}^{-1} \tag{2.30}$$

which they use to obtain the natural gradient updates

$$\tilde{\nabla}_\mu h = \boldsymbol{\Sigma} \nabla_\mu h \tag{2.31}$$

$$\tilde{\nabla}_\Sigma h = 2 \boldsymbol{\Sigma} \nabla_\Sigma h \boldsymbol{\Sigma} \tag{2.32}$$

By rewriting the above and inserting into the respective gradient expressions of the ELBO they get the natural gradient updates

$$\tilde{\nabla}_\mu \mathcal{L} = \boldsymbol{\Sigma} \mathbb{E}[\nabla_w \log p(\mathcal{D}|\boldsymbol{w}) + \lambda \nabla_w \log p(\boldsymbol{w})] \tag{2.33}$$

$$\tilde{\nabla}_\Sigma \mathcal{L} = -\mathbb{E}[\nabla^2_w \log p(\mathcal{D}|w) + \nabla^2_w \log p(\boldsymbol{w})] - \lambda \boldsymbol{\Sigma}^{-1} \tag{2.34}$$

where λ here is the KL weighting term (they consider they ELBO as $\mathcal{L} = \mathbb{E}_q[\log p(\mathcal{D}|\boldsymbol{w})] - \lambda \mathrm{KL}(q(\boldsymbol{w})||p(\boldsymbol{w}))$ where one can decide how much weight the KL term should have). Thus they can rewrite these updates in a way which resembles NGPE updates with correlated weight noise. We will not go into further details here but rather highlight some of the important parts of their derivations in the following.

A crucial step in their updates rely on Bonnet's and Price's theorem, which can be found in e.g. appendix A in [OA09], which allows them to swap the gradients with respect to (w.r.t.) variational parameters of expectations, with expectations of gradients w.r.t. the weights of the distributions. See section 2.5.3 for a more formal description. Thus, they are able to express natural gradients w.r.t. variational parameters with gradients w.r.t. weights, thereby removing the need for creating random variables of the variational parameters in the code.

Remark In terms of computational costs, using Bonnet's and Price's Theorem in Opper and Archambeau's derivation unlocks the ability to reduce the number of gradient computations by half. This is a direct result of being able to express the gradients w.r.t the variational probability distribution parameters with the gradients w.r.t. a realization of the weights through the reparameterization trick. Thus, since the gradients w.r.t the weights are obtained in the normal setup of existing deep learning codebases during backpropagation, we obtain the gradients w.r.t e.g. the variance parameter for free.

Another important step to consider in their derivations, is equation (8) in [Zha17] where they approximate the negative log-likelihood Hessian with the NGPE Fisher matrix. Specifically, $\nabla^2_{\boldsymbol{w}} \log p(y|\boldsymbol{x},\boldsymbol{w}) \approx \mathcal{D}\boldsymbol{w}\mathcal{D}\boldsymbol{w}^T := (\nabla_{\boldsymbol{w}} \log p(y|\boldsymbol{x},\boldsymbol{w}))^2$. Unfortunately, they do not go into much detail about this approximation, even though it holds crucial information on when the optimizer can be expected to behave appropriately. Thus, we draw on the theory presented in [Kha18], which presents a near identical noisy Adam method which they term Vadam. In [Kha18] the relationship between the Hessian and its approximation can be summarized as follows. Let $f_i(\boldsymbol{w})$ denote the negative log-likelihood of minibatch i and let $\mathcal{M}$ denote the set of minibatches. Then we have the following series of approximations

$$\nabla^2_{w_j w_j} f(\boldsymbol{w}) \approx \frac{1}{M}\sum_{i\in\mathcal{M}} [\nabla_{w_j} f_i(w)]^2 \approx \left[\frac{1}{M}\sum_{i\in\mathcal{M}} \nabla_{w_j} f_i(w)\right]^2 \tag{2.35}$$

where the first approximation follows from the Generalized Gauss-Newton approximation as given by Graves in [Gra11] and the second approximation follows from Gradient Magnitude [Bot16]. The reason for the second approximation, is due to the way current codebases compute the gradients, which compute the sum of the gradients over minibatches directly instead of computing the individual gradients [Kha18].

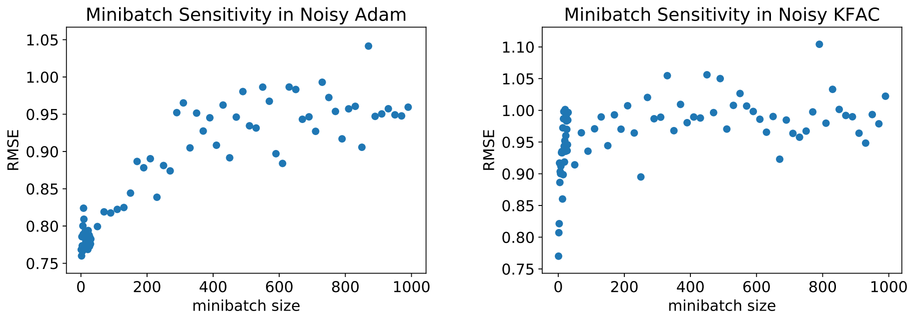


Figure 2.3: *The figures show how the performance of the algorithms deteriorate with increasing minibatch size for the wine dataset. This is possibly due to the fact that the Hessian approximation becomes distorted as minibatch size increases.* ***Left:*** *Noisy Adam's performance on the wine dataset gradually decreases with increasing minibatch size.* ***Right:*** *Noisy KFAC's performance rapidly deteriorates with increasing minibatch size.*

By utilizing the approximations in (2.35) we see that the larger the batch size, the less accurate the approximation to the Hessian becomes. For instance, in the case of batch-learning we would have no second order information. Conversely, doing online-learning would lead to an exact Generalized Gauss-Newton approximation. These considerations are also formalized in Theorem 1 in [Kha18]. In conclusion, one has to be mindful when choosing the batch-size, since this will affect how accurate the approximation to the Hessian becomes. Figure 2.3 tries to illustrate this relationship. It shows the Rooted Mean Squared Error (RMSE) loss over different minibatch sizes for two Bayesian neural networks, one using Noisy Adam and the other using Noisy KFAC (the models were trained on the Wine dataset which we present in chapter 4). Evidently, the performance decreases with increasing minibatch size which potentially can be atttributed to the loss of second order information.

NNG in itself does not resemble Adam, since the update rule does not have a momentum update term. This is perhaps one of the critiques about the way Zhang et al. derive their noisy Adam algorithm. Instead of having a sound theoretical basis for a momentum update, they *ad hoc* incorporate a momentum term in their NNG method to make it resemble Adam. Furthermore, their derivation of the natural gradient for a multivariate Gaussian as presented in appendix A of [Zha17] is not entirely accurate. Specifically, the way they obtain the Fisher matrix in eq. (16) in [Zha17] is flawed in the sense that they approximate the Fisher of the mean parameter $\boldsymbol{\mu}$ as, $\boldsymbol{F}_\mu = \nabla^2_\mu \mathrm{D}_{\mathrm{KL}} = \boldsymbol{\Sigma}_0^{-1} \approx \boldsymbol{\Sigma}^{-1}$. They should rather obtain natural gradient updates in natural parameter space and by the use of a so called mirror-descent update avoid direct computation of the Fisher matrix, as is done in [Kha18], which we will cover in section 2.5.2.

Lastly, it is worth mentioning that Noisy Adam does not take the square root of the Fisher approximation as is done in Adam, where they take the square root of the squared gradient.

```
**Algorithm 1** Noisy Adam
**Require:** $\alpha$: Stepsize
**Require:** $\beta_1$, $\beta_2$: Exponential decay rates for updating $\boldsymbol{\mu}$ and the Fisher $\boldsymbol{F}$
**Require:** $\lambda$, $\eta$, $\gamma_{\mathrm{ex}}$: KL weighting, prior variance, extrinsic damping term
$\boldsymbol{m} \leftarrow \mathbf{0}$
Calculate the intrinsic damping term $\gamma_{\mathrm{in}} = \frac{\lambda}{N\eta}$, total damping term $\gamma = \gamma_{\mathrm{in}} + \gamma_{\mathrm{ex}}$
**while** stopping criterion not met **do**
    $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{\mu}, \frac{\lambda}{N}\mathrm{diag}(\boldsymbol{f} + \gamma_{\mathrm{in}})^{-1})$
    $\boldsymbol{v} \leftarrow \nabla_w \log p(y|\boldsymbol{x}, \boldsymbol{w}) - \gamma_{\mathrm{in}} \cdot \boldsymbol{w}$
    $\boldsymbol{m} \leftarrow \beta_1 \cdot \boldsymbol{m} + (1 - \beta_1) \cdot \boldsymbol{v}$
    $\boldsymbol{f} \leftarrow \beta_2 \cdot \boldsymbol{f} + (1 - \beta_2) \cdot (\nabla_w \log p(y|\boldsymbol{x}, \boldsymbol{w}))^2$
    $\tilde{\boldsymbol{m}} \leftarrow \boldsymbol{m}/(1 - \beta_1^k)$
    $\hat{\boldsymbol{m}} \leftarrow \tilde{\boldsymbol{m}}/(\boldsymbol{f} + \gamma)$
    $\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} + \alpha \cdot \hat{\boldsymbol{m}}$
**end while**
```

Table 2.1: *Noisy Adam algorithm as presented in* [Zha17]. *Differences from Adam are highlighted in red.*

2.5.1.2 Noisy KFAC

Noisy KFAC is an optimizer which resembles Kronecker-Factored Approximate Curvature (KFAC) as proposed in [MG15]. In short, KFAC is a method for approximating natural gradient descent by approximating the Fisher matrix of the given model with a Kronecker product approximation. Thus, we move away from the restrictive fully factorized implementations presented in Noisy Adam and Vprop.

Noisy KFAC is derived through the use of the NNG method, extending it to Matrix Variate Gaussians (MVG). An MVG distribution is essentially a Normal distribution where the covariance is a Kronecker product of row-wise and column-wise covariances. We denote a random matrix $\mathbf{W} \in \mathbb{R}^{n \times p}$ as MVG distributed as follows: $\mathbf{W} \sim \mathcal{MN}_{n\times p}(\mathbf{W} \mid \mathbf{M}, \mathbf{A}, \mathbf{S})$ where $\mathbf{M}$ is the mean matrix, $\mathbf{A}$ the among-row covariance matrix and $\mathbf{S}$ the among-column covariance matrix. We then have $\text{vec}(\mathbf{W}) \sim \mathcal{N}_{np}(\text{vec}(\mathbf{M}), \mathbf{S} \otimes \mathbf{A})$.

For noisy KFAC, $\mathbf{A}$ and $\mathbf{S}$ are the covariance matrices of the input activations and pre-activation gradients, respectively. Figure 2.4 shows how a single neuron in a neural network looks. In the figure, the input is termed the input activations and the left half of the neuron is the pre-activation. Notation wise we have the pre-activation output as $\boldsymbol{s}_l = \boldsymbol{W}_l^T \boldsymbol{a}_l$ where $\boldsymbol{a}_l$ is the input for layer l and $\boldsymbol{W}_l$ is the weight matrix for layer l as defined in [Zha17], where the gradient of the pre-activation output gets defined as $\mathcal{D}\boldsymbol{s}_l = \nabla_{s_l} \log p(y|\boldsymbol{x}, \boldsymbol{w})$. These quantities, $\boldsymbol{a}_l$ and $\mathcal{D}\boldsymbol{s}_l$ are used for updating the matrices $\mathbf{A}$ and $\mathbf{S}$ in an exponential moving average fashion:

$$\mathbf{A}_l \leftarrow (1-\beta)\mathbf{A}_l + \beta \boldsymbol{a}_l \boldsymbol{a}_l^T \tag{2.36}$$

$$\mathbf{S}_l \leftarrow (1-\beta)\mathbf{S}_l + \beta \mathcal{D}\boldsymbol{s}_l \mathcal{D}\boldsymbol{s}_l^T \tag{2.37}$$

Thus, noisy KFAC is obtained through an adaptation of the proposed NNG method to the KFAC framework.

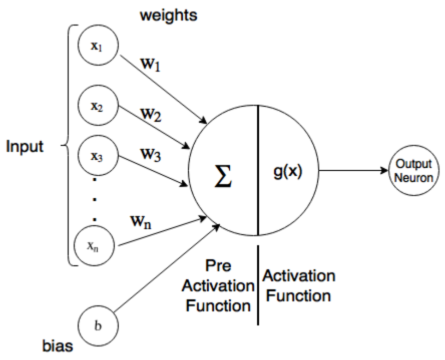


Figure 2.4: *An artifical neuron: Basic unit of Neural Networks. From [18b]*

Algorithm 2 Noisy K-FAC. Subscript l denotes layers, $\boldsymbol{w}_l = \mathrm{vec}(\boldsymbol{W}_l)$, and $\boldsymbol{\mu}_l = \mathrm{vec}(\boldsymbol{M}_l)$. We assume zero momentum for simplicity.

```
Require: $\alpha$: Stepsize
Require: $\beta$: exponential moving average parameter
Require: $\lambda$, $\eta$, $\gamma_{\mathrm{ex}}$: KL weighting, prior variance, extrinsic damping term
Require: stats and inverse update intervals $T_{\mathrm{stats}}$ and $T_{\mathrm{inv}}$
$k \leftarrow 0$ and initialize $\{\boldsymbol{\mu}_l\}_{l=1}^{L}$, $\{\boldsymbol{S}_l\}_{l=1}^{L}$, $\{\boldsymbol{A}_l\}_{l=1}^{L}$
Calculate the intrinsic damping term $\gamma_{\mathrm{in}} = \frac{\lambda}{N\eta}$, total damping term $\gamma = \gamma_{\mathrm{in}} + \gamma_{\mathrm{ex}}$
while stopping criterion not met do
    $k \leftarrow k + 1$
    $\boldsymbol{W}_l \sim \mathcal{MN}(\boldsymbol{M}_l, \frac{\lambda}{N}[\boldsymbol{A}_l^{\gamma_{\mathrm{in}}}]^{-1}, [\boldsymbol{S}_l^{\gamma_{\mathrm{in}}}]^{-1})$
    if $k \equiv 0 \pmod{T_{\mathrm{stats}}}$ then
        Update the factors $\{\boldsymbol{S}_l\}_{l=1}^{L}$, $\{\boldsymbol{A}_l\}_{l=0}^{L-1}$ using eq. (12)
    end if
    if $k \equiv 0 \pmod{T_{\mathrm{inv}}}$ then
        Calculate the inverses $\{[\boldsymbol{S}_l^{\gamma}]^{-1}\}_{l=1}^{L}$, $\{[\boldsymbol{A}_l^{\gamma}]\}_{l=0}^{L-1}$ using eq. (13)
    end if
    $\boldsymbol{V}_l = \nabla_{w_l} \log p(y|\boldsymbol{x}, \boldsymbol{w}) - \gamma_{\mathrm{in}} \cdot \boldsymbol{W}_l$
    $\boldsymbol{M}_l \leftarrow \boldsymbol{M}_l + \alpha[\boldsymbol{A}_l^{\gamma}]^{-1}\boldsymbol{V}_l\,[\boldsymbol{S}_l^{\gamma}]^{-1}$
end while
```

Table 2.2: *Noisy KFAC algorithm as presented in* [Zha17]. *Differences from KFAC are highlighted in red.*

2.5.2 Vprop: Variational Inference using RMSprop

In this section we cover the Vprop optimizer which closely resembles RMSprop, originally proposed by George Hinton in [Hin]. Like Adam, RMSprop is an optimizer which, contrary to e.g. Stochastic Gradient Descent (SGD), employs an adaptive learning rate scheme where the learning rate is iteratively updated with a moving average of the squared gradient.

Vprop was originally proposed in [Kha17]. However, late in the stage of this thesis, a modified version of Vprop has been proposed by the same author(s) in [Kha18]. The modified version of Vprop enables learning on larger datasets such as MNIST, than what was proposed in the original paper. Thus, in this thesis we consider the latest version of Vprop. Both the original and modified Vprop algorithm is presented in table 2.3.

Unlike NNG, Vprop derives natural gradient updates through a connection between a mirror descent update in *expectation parameter space* and a natural gradient descent update in *natural parameter space*. The connection being, that the two are equivalent. Recall, for an exponential family, η denotes the natural parameter,

$\varphi(x)$ denotes the sufficient statistics and m denotes the expectation parameter, i.e. $m = \mathbb{E}(\varphi(\eta))$. Then, using the definition from [Kha18], a mirror descent step is defined as

$$\boldsymbol{m}_{t+1} = \arg\min_{m} \langle \boldsymbol{m}, -\nabla_m \mathcal{L}_*(\boldsymbol{m}_t) \rangle + \frac{1}{\beta_t} \mathrm{KL}[q_m(\boldsymbol{\theta}) || q_{m_t}(\boldsymbol{\theta})] \tag{2.38}$$

Note here that $\mathcal{L}_*$ denotes the ELBO in terms of the expectation parameter $\boldsymbol{m}$ and $\langle \cdot \,, \cdot \rangle$ denotes inner product. Furthermore, recall that a natural gradient descent step is defined as

$$\boldsymbol{\eta}_{t+1} = \boldsymbol{\eta}_t + \beta_t \boldsymbol{F}(\boldsymbol{\eta}_t)^{-1} \nabla_\eta \mathcal{L}(\boldsymbol{\eta}_t) \tag{2.39}$$

where the step now is taken in natural parameter space. As mentioned, the mirror descent update and the natural gradient update above are equivalent (see e.g. theorem 2 in [Kha18]). They use this equivalence together with a result which states that the natural gradient with respect to a natural parameter is equivalent to the gradient with respect to an expectation parameter, i.e. $\boldsymbol{F}(\boldsymbol{\eta})^{-1} \nabla_\eta \mathcal{L}(\boldsymbol{\eta}) = \nabla_m \mathcal{L}_*(\boldsymbol{m})$ (See [Kha18] for more details). These properties, enable them to write the natural gradient update as follows: $\boldsymbol{\eta}_{t+1} = \boldsymbol{\eta}_t + \beta_t \nabla_m \mathcal{L}_*(\boldsymbol{m}_t)$. This way, they avoid having to directly compute the Fisher matrix in the natural gradient update. Next, by considering a Normal distribution, they can rewrite the gradient with respect to the expectation parameter, as a gradient with respect to the variational parameters $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ and this way obtain natural gradient updates for distribution parameters. Thus they arrive at the natural gradient updates (see [Kha18] or [Kha17] for a full derivation)

$$\boldsymbol{\mu}_{t+1} = \boldsymbol{\mu}_t + \beta_t \boldsymbol{\sigma}^2_{t+1} \circ [\nabla_\mu \mathcal{L}_t] \tag{2.40}$$

$$\boldsymbol{\sigma}^{-2}_{t+1} = \boldsymbol{\sigma}_t^{-2} - 2\beta_t [\nabla_{\sigma^2} \mathcal{L}_t] \tag{2.41}$$

By using some of the same tricks and derivations as for NNG, they obtain the Vprop algorithm as presented in table 2.3. Specifically, they rely on the swapping of gradients of expectations with expectations of gradients as detailed in section 2.5.3. Furthermore, they also rely on the gradient magnitude approximation to the Hessian. Lastly, in contrast to Noisy Adam, they still take the square root of the squared gradient in Vprop.

On a side note, the Variational Adam (Vadam) proposed in the same paper, is more in the spirit of Adam, when compared to noisy Adam, since they have a better motivation for adding momentum and they keep the square root on the Fisher approximation.

Algorithm 3 Vprop

```
$\boldsymbol{w} \leftarrow \boldsymbol{\mu} + \boldsymbol{\varepsilon}./\sqrt{\boldsymbol{s} + \lambda}$
$\boldsymbol{g} \leftarrow \nabla_w f(\boldsymbol{w})$
$\boldsymbol{s} \leftarrow (1-\beta)\boldsymbol{s} + \beta(\boldsymbol{g}. * \boldsymbol{g})$
$\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} - \alpha((\boldsymbol{g} + \lambda\boldsymbol{\mu})./(\boldsymbol{s} + \lambda))$
```

Algorithm 4 Modified Vprop

```
$\boldsymbol{w} \leftarrow \boldsymbol{\mu} + \boldsymbol{\varepsilon}./\sqrt{N\boldsymbol{s} + \lambda}$
$\boldsymbol{g} \leftarrow \nabla_w f(\boldsymbol{w})$
$\boldsymbol{s} \leftarrow (1-\beta)\boldsymbol{s} + \beta(\boldsymbol{g}. * \boldsymbol{g})$
$\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} - \alpha((\boldsymbol{g} + \frac{\lambda}{N}\boldsymbol{\mu})./(\sqrt{\boldsymbol{s}} + \frac{\lambda}{N}))$
```

Table 2.3: *Vprop (left) and modified Vprop (right) algorithms as presented in* [Kha17] *and* [Kha18]. *Differences from RMSprop are highlighted in red.*

2.5.3 Closing notes on the optimizers

Before leaving this section we highlight some of the commonalities between the different optimizers.

The optimizers all make use of the reparameterization trick to obtain a noisy sample of weights, i.e. $\boldsymbol{w} = \boldsymbol{\mu} + \boldsymbol{\sigma} \circ \boldsymbol{\varepsilon}$. The gradient obtained is then with respect to this noisy sample which is being used in the update scheme.

Furthermore, the number of examples (N in the algorithms presented) which are being used for damping the updates is the total number of training examples in a training set. This follows because both papers assume that the likelihood is on the form $p(\mathcal{D}|\boldsymbol{w}) := \frac{1}{N}\sum_i^N \log p(y_i|X_i, w)$ (see e.g. equation (6) in [Kha18]).

A crucial common point in the derivations in both papers, is the swapping of gradients of expectations, with expectations of gradients. This allows them to take gradients with respect to weights instead of variational parameters. Specifically, they rely on Bonnet's and Price's theorem which state that when considering expectations of any function f with a Normal distribution as the probability measure, then the following identities hold

$$\nabla_\mu \mathbb{E}_{\mathcal{N}(\mu,\Sigma)}[f(w)] = \mathbb{E}_{\mathcal{N}(\mu,\Sigma)}[\nabla_w f(w)] \tag{2.42}$$

$$\nabla_\Sigma \mathbb{E}_{\mathcal{N}(\mu,\Sigma)}[f(w)] = \frac{1}{2}\mathbb{E}_{\mathcal{N}(\mu,\Sigma)}[\nabla_w^2 f(w)] \tag{2.43}$$

As stated above, this is one of the keys to move from natural gradient updates where the gradient is with respect to variational parameters, to gradients with respect to the weights.

This concludes our review of the optimizers performing variational inference.

CHAPTER 3

Hierarchical Models

So far, we have focused primarily on Gaussian mean-field models which use a perhaps overly simplistic choice for a prior distribution. In this chapter we consider extending these to employ a hierarchical structure. By hierarchical we mean placing a prior distribution on a parameter of our Gaussian distribution (a hyperprior), e.g. placing a distribution on the variance parameter.

As an illustrative first example, we consider a simple hierarchical structure which could be an inverse gamma distribution placed on the variance parameter of a Normal distribution. This leads to a scenario where every single weight has its own variance, drawn from a gamma distribution (instead of every single weight having its own variance, one could also assume that e.g. blocks of weights would have each their own variance parameter from block to block, i.e. group sparsity). Figure 3.1 tries to illustrate the situation. Figure 3.1a shows the complete independence between the weights, with λ_i being the variance parameter. For the models we have discussed so far, the prior variance is the same for all weights such that $\lambda_1 = \lambda_2 = \cdots = \lambda_n$. Figure 3.1b shows how placing a prior with hyperparameters $\boldsymbol{\theta} = \{\alpha, \beta\}$ on the variance changes the model such that each variance parameter now comes from a prior distribution. We formalize the setup mentioned above as follows:

$$\boldsymbol{w}|\boldsymbol{\lambda} \sim \mathcal{N}(\boldsymbol{w}|\mathbf{0}, \boldsymbol{\lambda}^{-1}), \quad \boldsymbol{\lambda} \sim \Gamma(\boldsymbol{\lambda}|\alpha, \beta) \tag{3.1}$$

where $\boldsymbol{w}$ are the weights and $\boldsymbol{\lambda}$ are the Gamma distributed random variables. Thus the prior becomes $p(\boldsymbol{w}, \boldsymbol{\lambda}) = p(\boldsymbol{w}|\boldsymbol{\lambda})p(\boldsymbol{\lambda})$. This joint distribution compounds to a student's t distribution. To realize this, we for simplicity consider a univariate case, though the result still holds for the multivariate case since we assume that the distribution factorizes as $\log p(\boldsymbol{w}, \boldsymbol{\lambda}) = \log \prod_i p(w_i, \lambda_i)$. We can marginalize out λ

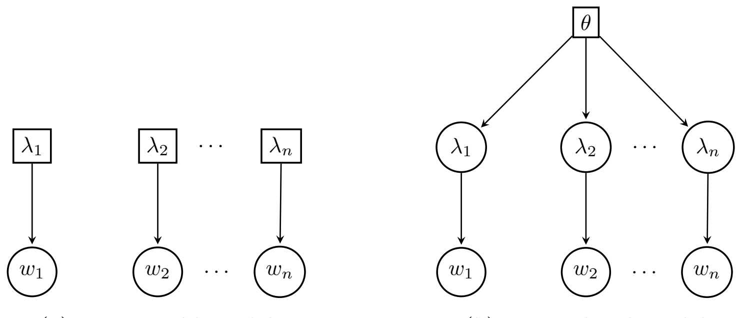
from the joint distribution to get $p(w)$ as follows:

$$\begin{aligned} p(w) &= \int_0^\infty p(w|\lambda^{-1})p(\lambda)d\lambda && (3.2)\\ &\propto \int_0^\infty \sqrt{\lambda}\exp\left(-\lambda\frac{(w-\mu)^2}{2}\right)\lambda^{a-1}\exp\left(-\frac{\lambda}{b}\right)d\lambda && (3.3)\\ &= \int_0^\infty \lambda^{a-\frac{1}{2}}\exp\left(-\lambda\left(\frac{(w-\mu)^2}{2}+\frac{1}{b}\right)\right)d\lambda && (3.4)\\ &= \int_0^\infty \lambda^{a-\frac{1}{2}}\exp\left(-\lambda\frac{(w-\mu)^2+2}{2b}\right)d\lambda && (3.5) \end{aligned}$$

Now substitute with $z = \lambda\frac{(w-\mu)^2-2}{2b}$ and differentiate such that $d\lambda = \left(\frac{(w-\mu)^2-2}{2b}\right)^{-1}dz$. Inserting this expression in the integral above yields

$$\begin{aligned} p(w) &= \int_0^\infty \left(\frac{(w-\mu)^2+2}{2b}\right)^{-1}\lambda^{a-\frac{1}{2}}e^{-z}dz && (3.6)\\ &= \left(\frac{(w-\mu)^2+2}{2b}\right)^{-(a+\frac{1}{2})}\int_0^\infty \left(\frac{(w-\mu)^2+2}{2b}\right)^{a-\frac{1}{2}}\lambda^{a-\frac{1}{2}}e^{-z}dz && (3.7)\\ &= \left(\frac{(w-\mu)^2+2}{2b}\right)^{-(a+\frac{1}{2})}\Gamma\left(a+\frac{1}{2}\right) && (3.8) \end{aligned}$$

Now, since the gamma function evaluates to a constant we see that we are left with an expression which is proportional to the student's t distribution with $\nu = 2a$ degrees of freedom, location $l = \mu$ and scale $s = \sqrt{ab}$ as desired.



(a) Mean-Field model (b) Hierarchical model

Figure 3.1: ***(a)*** *shows the typical model configuration for a mean-field model where each weight has its own independent prior variance which typically is chosen to be the same for all weights.* ***(b)*** *shows an example of a hierarchical model where the variance parameter now comes from a distribution, thus becoming a random variable itself.*

The student's t distribution offers fatter tails than the Normal distribution and a sharper concentration around zero. Figure 3.2 illustrates this by showing a standard Normal distribution and a student's t distribution for various degrees of freedom. It is a well known fact that the student's t distribution approaches a Normal distribution as the degrees of freedom goes to infinity. Conversely, as the degrees of freedom decrease, the tails get increasingly fatter, which allows strong signals learned to remain large as figure 3.2 illustrates. This is one of the benefits of the student's t distribution as opposed to a Normal distribution when considering the choice of a prior. Later we present experiments and benchmarks on a model which uses the student's t distribution as a prior as well as variational posterior, in comparison with the Normal distribution.

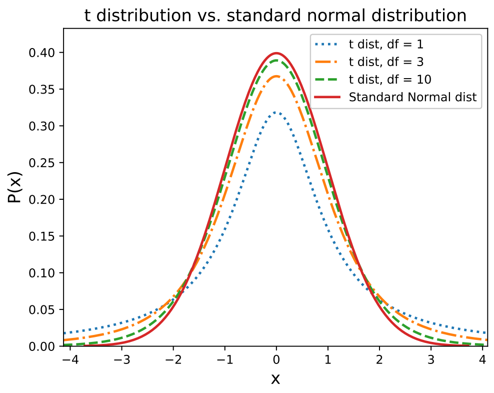


Figure 3.2: *Comparsion between a student's t distribution and a standard Normal distribution. Note the convergence towards the Normal distribution as degrees of freedom increases.*

3.1 The Horseshoe prior

The example with the student's t distribution above is perhaps too simple as an illustration of hierarchical models. Since we marginalize out the distribution of the variance parameters we are in a way "removing" the hierarchy of the model. Instead we can consider another sparsity inducing distribution, the Horseshoe. For the Horseshoe, the setup is as follows

$$\boldsymbol{w}|\tau, \boldsymbol{\lambda} \sim \mathcal{N}(0, \tau^2\boldsymbol{\lambda}^2), \quad \tau \sim C^+(0, \kappa), \quad \boldsymbol{\lambda} \sim C^+(0, 1) \tag{3.9}$$

Here C^+ denotes the Half-Cauchy distribution which is simply the positive half of the Cauchy distribution (see appendix C for density plots of the two). Note here that τ is a single random variable where $\boldsymbol{\lambda}$ is of same dimensionality of $\boldsymbol{w}$. Thus τ is multiplied elementwise on to each λ_i. This is the unique part about the Horseshoe. τ and $\boldsymbol{\lambda}$ are what is typically called the global and local variance parameters, respectively. The interpretation being that the global variance parameter enforces shrinking of all the weights while the local variance parameter allows weights with strong signals to remain large. Figure 3.3 shows the density of the Horseshoe in comparison with the student's t distribution. The Horseshoe has an infinitely tall spike at 0 which enables shrinkage of weights to essentially be 0 unlike e.g. the usual Normal distribution. Also note that the only parameter we can choose when defining our prior is κ which determines the shrinkage effect. The smaller κ is, the sharper the spike at zero. So we determine the prior global variance parameter κ dependent on e.g. our beliefs about the data.

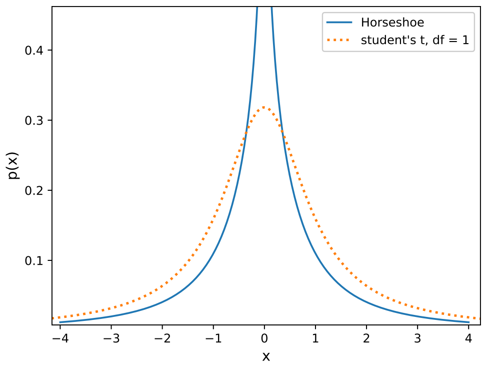


Figure 3.3: *Density of the Horseshoe distribution and the student's t distribution with 1 degree of freedom. Note the infinitely tall spike at 0 for the Horseshoe, while still having fat tails.*

3.1.1 Horseshoe details and ELBO

The Horseshoe setup in (3.9) is a generalization of the Horseshoe and does not show the intricacies which we now highlight. In the Horseshoe model we employ group sparsity, meaning that we consider individual local variance parameters for each *hidden unit* in the network as is also done in [GD17]. Thus by denoting $\boldsymbol{w}_i$ as the weight vector for hidden unit i we have for the weights of a single hidden unit i

$$\boldsymbol{w}_i|\tau, \lambda_i \sim \mathcal{N}(0, \tau^2\lambda_i^2\mathbf{I}), \quad \tau \sim C^+(0, \kappa), \quad \lambda_i \sim C^+(0, 1) \tag{3.10}$$

Now according to [GD17], because the weights in (3.10) are heavily correlated with the variance parameters, it can be difficult to approximate the weights. A way to solve this is by considering a non-centered parameterization of the weights given by $\boldsymbol{w}_i = \tau\lambda_i\widetilde{\boldsymbol{w}}_i$ where $\widetilde{\boldsymbol{w}}_i \sim \mathcal{N}(0, \mathbf{I})$. This parameterization ensures marginal correlation between weights and variance parameters and enables us to better find sparse solutions [GD17].

Certain problems exist with using Cauchy distributions as priors and/or variational posteriors explicitly. One problem being that exponential family distributions struggle to capture the long flat Cauchy tails. Another problem being that using Cauchy distributions as approximating posterior distributions can lead to high variance gradients [GD17]. To resolve this, we follow [NOW14] (similarly we could instead follow [GD17] or [LUW17]) and write up the Horseshoe in a parameterized fashion using Inverse-Gamma and Gamma distributions.

For the following setup let $\Gamma(\alpha, \beta)$ denote the gamma distribution and let $\Gamma^{-1}(\alpha, \beta)$ denote the inverse gamma distribution. Then equation (3.10) is equivalent to the following:

$$\overbrace{\tau_a \sim \Gamma\left(\frac{1}{2}, \kappa^2\right), \quad \tau_b \sim \Gamma^{-1}\left(\frac{1}{2}, 1\right)}^{\text{Global variance}}, \quad \overbrace{\lambda_{a,i} \sim \Gamma\left(\frac{1}{2}, 1\right), \quad \lambda_{b,i} \sim \Gamma^{-1}\left(\frac{1}{2}, 1\right)}^{\text{Local variance}} \tag{3.11}$$

$$\tilde{\boldsymbol{w}}_i \sim \mathcal{N}(0, \mathbf{I}), \qquad \boldsymbol{w}_i = \tilde{\boldsymbol{w}}_i \sqrt{\tau_a \tau_b \lambda_{a,i} \lambda_{b,i}} \tag{3.12}$$

The prior then becomes (shown here for a single hidden layer with parameters $\boldsymbol{\theta} = \{\tau_a, \tau_b, \boldsymbol{\lambda}_a, \boldsymbol{\lambda}_b\}$)

$$p(\boldsymbol{\theta}) = p(\tau_a, \tau_b, \boldsymbol{\lambda}_a, \boldsymbol{\lambda}_b) = p(\tau_a)p(\tau_b)p(\boldsymbol{\lambda}_a)p(\boldsymbol{\lambda}_b) \tag{3.13}$$

$$= \Gamma\left(\tau_a \middle| \frac{1}{2}, \kappa^2\right) \Gamma^{-1}\left(\tau_b \middle| \frac{1}{2}, 1\right) \prod_i^M \Gamma\left(\lambda_{a,i} \middle| \frac{1}{2}, 1\right) \prod_i^M \Gamma^{-1}\left(\lambda_{b,i} \middle| \frac{1}{2}, 1\right) \tag{3.14}$$

where M is the total number of hidden units in the layer.

For ease of computation, the variational posterior is chosen such that the KL divergence between the prior and variational posterior can be expressed in closed form. Thus, we approximate the gamma and inverse gamma distributions with fully factorized log-Normal distributions, which we denote as $L\mathcal{N}$. For the prior parameters $\boldsymbol{\theta} = \{\tau_a, \tau_b, \boldsymbol{\lambda}_a, \boldsymbol{\lambda}_b\}$ we then have the variational posterior (shown here for a single hidden layer)

$$q(\boldsymbol{\theta}|\boldsymbol{\phi}) = q(\tau_a, \tau_b, \boldsymbol{\lambda}_a, \boldsymbol{\lambda}_b|\boldsymbol{\phi}) \tag{3.15}$$

$$q(\tau_a|\mu_{\tau_a}, \sigma^2_{\tau_a})q(\tau_b|\mu_{\tau_b}, \sigma^2_{\tau_b}) \tag{3.16}$$

$$q(\boldsymbol{\lambda}_a|\boldsymbol{\mu}_{\lambda_a}, \boldsymbol{\sigma}^2_{\lambda_a})q(\boldsymbol{\lambda}_b|\boldsymbol{\mu}_{\lambda_b}, \boldsymbol{\sigma}^2_{\lambda_b}) \tag{3.17}$$

$$= L\mathcal{N}(\tau_a|\mu_{\tau_a}, \sigma^2_{\tau_a})L\mathcal{N}(\tau_b|\mu_{\tau_b}, \sigma^2_{\tau_b}) \tag{3.18}$$

$$\prod_i^M L\mathcal{N}(\lambda_{a,i}|\mu_{\lambda_a,i}, \sigma^2_{\lambda_a,i}) \prod_i^M L\mathcal{N}(\lambda_{b,i}|\mu_{\lambda_b,i}, \sigma^2_{\lambda_b,i}) \tag{3.19}$$

with $\boldsymbol{\phi} = \{\mu_{\tau_a}, \sigma^2_{\tau_a}, \mu_{\tau_b}, \sigma^2_{\tau_b}, \boldsymbol{\mu}_{\lambda_a}, \boldsymbol{\sigma}^2_{\lambda_a}, \boldsymbol{\mu}_{\lambda_b}, \boldsymbol{\sigma}^2_{\lambda_b}\}$. Lastly, we have the prior and variational posterior of the non-centered weights $p(\widetilde{\boldsymbol{w}}) = \mathcal{N}(\widetilde{\boldsymbol{w}}|0, \mathbf{I})$ and $q(\widetilde{\boldsymbol{w}}|\boldsymbol{\mu}_w, \text{diag}(\Sigma_w))$ respectively.

Because we consider a fully factorized model, we can separate the KL divergence of the joint distribution into the KL divergence between the individual parameters. Thus we can write the ELBO as (again shown for a single hidden layer to simplify notation)

$$\mathcal{L}(\boldsymbol{\phi}) = -\mathbb{E}_q(\log p(\mathcal{D}|\boldsymbol{w})) + \text{KL}(q(\tilde{\boldsymbol{w}}, \boldsymbol{\theta})||p(\tilde{\boldsymbol{w}}, \boldsymbol{\theta})) \tag{3.20}$$

$$= -\sum_n^N \mathbb{E}_q(\log p(y_n|\boldsymbol{w}, \boldsymbol{x}_n)) + \text{KL}(q(\tau_a)||p(\tau_a)) + \text{KL}(q(\tau_b)||p(\tau_b)) \tag{3.21}$$

$$+ \prod_i^M \text{KL}(q(\lambda_{a,i})||p(\lambda_{a,i})) + \prod_i^M \text{KL}(q(\lambda_{b,i})||p(\lambda_{b,i})) + \text{KL}(q(\tilde{\boldsymbol{w}})||p(\tilde{\boldsymbol{w}})) \tag{3.22}$$

where again, M is the total number of hidden units in the given layer. Following the derivations in [LUW17] we can write out the KL divergence terms in closed form

$$-\text{KL}(q(\tau_a)||p(\tau_a)) = \log\kappa - \frac{1}{\kappa}\exp\left(\mu_{\tau_a} + \frac{1}{2}\sigma^2_{\tau_a}\right) + \frac{1}{2}(\mu_{\tau_a} + \log\sigma^2_{\tau_a} + 1 + \log 2) \tag{3.23}$$

$$-\text{KL}(q(\tau_b)||p(\tau_b)) = -\exp\left(\frac{1}{2}\sigma^2_{\tau_b} - \mu_{\tau_b}\right) + \frac{1}{2}(-\mu_{\tau_b} + \log\sigma^2_{\tau_b} + 1 + \log 2) \tag{3.24}$$

$$-\text{KL}(q(\boldsymbol{\lambda}_a||p(\boldsymbol{\lambda}_a)) = \sum_i^M \left(-\exp(\mu_{\lambda_a,i} + \frac{1}{2}\sigma^2_{\lambda_a,i}) + \frac{1}{2}(\mu_{\lambda_a,i} + \log\sigma^2_{\lambda_a,i} + 1 + \log 2)\right) \tag{3.25}$$

$$-\text{KL}(q(\boldsymbol{\lambda}_b||p(\boldsymbol{\lambda}_b)) = \sum_i^M \left(-\exp(\mu_{\lambda_b,i} + \frac{1}{2}\sigma^2_{\lambda_b,i}) + \frac{1}{2}(\mu_{\lambda_b,i} + \log\sigma^2_{\lambda_a,i} + 1 + \log 2)\right) \tag{3.26}$$

For the KL divergence of the non-centered weights $\widetilde{\boldsymbol{w}}$ we have that

$$\text{KL}(q(\tilde{\boldsymbol{w}})||p(\tilde{\boldsymbol{w}})) = \frac{1}{2}\sum_i^N \left(-\log\sigma^2_{w,i} + \sigma^2_{w,i} + \mu^2_{w,i} - 1\right) \tag{3.27}$$

where N is the total number of weights in a hidden layer. To realize this, we simply compute the KL divergence explicitly (here shown for a univariate case to ease notation)

$$\begin{aligned}
\mathrm{KL}(q(\widetilde{w})||p(\widetilde{w})) &= \mathbb{E}_q \log q(\widetilde{w}) - \mathbb{E}_q \log p(\widetilde{w}) && (3.28)\\
&= \mathbb{E}_q \left(-\frac{1}{2}\log(2\pi) - \log \sigma_w - \frac{1}{2}\sigma_w^{-2}(\widetilde{w}-\mu_w)^2\right) && (3.29)\\
&\quad - \mathbb{E}_q \left(-\frac{1}{2}\log(2\pi) - \frac{1}{2}\widetilde{w}^2\right) && (3.30)\\
&= -\log \sigma_w - \frac{1}{2}\sigma_w^{-2}(\mathbb{E}_q(\widetilde{w}^2) + \mu_w^2 - 2\mu_w \mathbb{E}_q(\widetilde{w})) + \frac{1}{2}\mathbb{E}_q(w^2) && (3.31)\\
&= -\log \sigma_w - \frac{1}{2} + \frac{1}{2}\sigma_w^2 + \frac{1}{2}\mu_w^2 && (3.32)\\
&= \frac{1}{2}\left(-\log \sigma_w^2 + \sigma_w^2 + \mu_w^2 - 1\right) && (3.33)
\end{aligned}$$

where we in (3.31) have used that $\mathbb{E}(w^2) = V(w) + (\mathbb{E}(w))^2 = \sigma^2 + \mu^2$. Thus the result in (3.27) is obtained by summing over all the weights as desired. Thus we have defined the ELBO of the Horseshoe model which can readily be coded in practice, following the above.

Another benefit of using log-Normal distributions in the variational posterior is that we easily can sample from these through the reparameterization trick, since we can write a log-Normal distributed variable x as $x = \exp(\mu + \sigma\varepsilon)$ with $\varepsilon \sim \mathcal{N}(0,1)$. This is useful when doing a forward pass through the network. To complete a forward pass we rely on ancestral sampling, which essentially means that we start by sampling from the top of the hierarchy and work our way down through the hierarchy, sampling when needed. Specifically, we follow along the lines of [LUW17] which summarizes the forward pass in a nice way (see table C.1 in appendix C for details).

As a closing note on the Horseshoe, it is worth taking a look at the shrinkage profile of the Horseshoe which is discussed in great detail in [MGG09]. Consider the local variance parameter $\lambda_i \sim C^+(0,1)$. By transformation of a random variable, we have that $z = \frac{1}{1+\lambda_i^2}$ becomes a random shrinkage coefficient which is Beta distributed, $z \sim \mathrm{Be}(1/2, 1/2)$ [MGG09]. The density is shown in figure 3.4 and reveals where the Horseshoe gets its name from. The density places most of its probability mass around 0 and 1 which is what enables both shrinkage and strong signals, since if λ_i is very large, z is close to zero and if λ_i is very small, z is close to 1. As mentioned in [MGG09], no other commonly used shrinkage prior exhibits these properties.

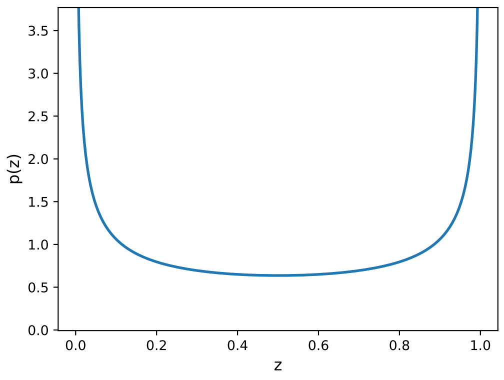


Figure 3.4: *Density of the random shrinkage coefficient* $z = \frac{1}{1+\lambda_i^2}$ where $\lambda_i \sim C^+(0,1)$.

3.2 Hierarchical noisy optimizer

A significant amount of time was devoted to investigating the possibility of extending noisy Adam to a hierarchical model. In this section we discuss the challenges we faced when trying to derive a hierarchical noisy Adam.

The overarching challenge with extending to hierarchical noisy optimizers is that the noisy optimizers have been specifically developed for Gaussian mean-field variational inference. Thus, they rely on certain methods and theorems which are only applicable to Gaussian distributions. The challenge then becomes, either to develop ones own hierarchical noisy optimizer by starting from the same origin as the original papers, i.e. the natural gradient framework; or try to argue for the case that the hierarchical model can be reformulated or in some way can be reduced to a Gaussian distribution.

Initially, to keep things simple, we started by investigating the case which we presented in equation (3.1). One of the first things we tried was to get natural gradient updates for the gamma distribution via the trust region optimization problem [Zha17] which we mentioned in section 2.5. Thus we computed the KL divergence for the gamma distribution p in some neighborhood of p_0:

$$\mathrm{KL}(\Gamma||\Gamma_0) = \mathbb{E}(\log p(\boldsymbol{w}) - \log p_0(\boldsymbol{w})) = \sum_i \mathbb{E}(\log p(w_i) - \log p_0(w_i)) \tag{3.34}$$

$$= \sum_i \left[(\alpha_i^{p_0} - \alpha_i^p)\psi(\alpha_i^{p_0}) + \log\left(\frac{\alpha_i^p}{\alpha_i^{p_0}}\right) + \alpha_i^p \left(\log\left(\frac{\beta_i^{p_0}}{\beta_i^p}\right)\right) + \alpha_i^{p_0} \frac{\beta_i^p - \beta_i^{p_0}}{\beta_i^{p_0}} \right] \tag{3.35}$$

Next we found the second order derivatives of the above to be

$$\nabla^2_\alpha \mathrm{KL} = \psi'(\boldsymbol{\alpha}_p) \tag{3.36}$$

$$\nabla^2_\beta \mathrm{KL} = \frac{\boldsymbol{\alpha}_p}{\boldsymbol{\alpha}_p^2} \tag{3.37}$$

where ψ' is the first order derivative of the polygamma function and operations here are done elementwise. Thus, we have the natural gradient updates

$$\tilde{\nabla}_\alpha h = \psi'(\boldsymbol{\alpha})\nabla_\alpha h \tag{3.38}$$

$$\tilde{\nabla}_\beta h = \boldsymbol{\alpha}\boldsymbol{\beta}^{-2}\nabla_\beta h \tag{3.39}$$

where h is some differentiable function such as e.g. the ELBO. The problem now becomes arguing for the case that you can interchange gradients and expectations in the case of a gamma distribution. Since Bonnet's and Price's theorem only holds for the case of a Normal distribution we considered different ways of achieving this for a gamma distribution. Methods such as those presented in Implicit Reparameterization Gradients [FMM18], Rejection Sampling Variational Inference [Nae+17] and Stochastic gradient variational Bayes for gamma approximating distributions [Kno15] (the latter being used in `PyTorch`), in effect extend the reparameterization trick to other distributions. However, the proposed methods in these papers makes use of more complicated machinery than a simple coordinate transform, since it is no easy matter to sample gamma distributed variables through the reparameterization trick. We did attempt to use the latter method without success.

Our initial approach perhaps lacks a proper theoretical base. Since we in a way tried to copy the derivations in [Zha17] and apply them to a gamma distribution, we end up doing pattern matching throughout the derivations in their paper instead of starting from where the theory originates.

Late in the progress of this thesis the paper Fast and Scalable Bayesian Deep Learning by Weight-Perturbation in Adam [Kha18] was submitted. As mentioned in chapter 2 this paper proposes an updated version of Vprop as well as another noisy Adam optimizer which they call Vadam. The paper has a more sound theoretical approach to developing the noisy optimizers they present. The paper heightened our understanding of how one obtains a noisy optimizer which resembles e.g. Adam. We also had a brief discussion with the author after a presentation of his paper where we discussed how one would extend the optimizer to a hierarchical structure. He proposed starting from the origin, which in this case is writing up the natural gradient updates in natural parameter space. The natural parameters for this specific distribution are (see [18a])

$$\eta^{(1)} = \alpha - \frac{1}{2}, \quad \eta^{(2)} = -\beta - \frac{1}{2}\lambda\mu^2, \quad \eta^{(3)} = \mu, \tag{3.40}$$

and the natural gradient update for each natural parameter becomes

$$\eta_{t+1} = \eta_t + \beta_t \boldsymbol{F}(\eta_t)^{-1} \nabla_\eta \mathcal{L}(\eta_t) \tag{3.41}$$

Using theorem 2 in [Kha18] this is equivalent to

$$\eta_{t+1} = \eta_t + \beta_t \nabla_m \mathcal{L}_*(m_t) \tag{3.42}$$

where m is the expectation parameter for a given natural parameter and $\mathcal{L}_*$ is now the ELBO but written in terms of the expectation parameter as previously mentioned. The first obstacle comes when we want to express $\nabla_m \mathcal{L}_*$ in terms of the normal parameters μ, α and β. For the Gaussian case we can rewrite it in a nice form using the chain rule [Kha18]. For instance, for the expectation parameter for the first sufficient statistic x of a Normal distribution, they obtain $\nabla_m \mathcal{L}_* = \nabla_\mu \mathcal{L} - 2\nabla_\Sigma \mathcal{L}\mu$. However, for this given hierarchical model, no such nice form exists (not one we could find at least).

If we assume that we manage to write $\nabla_m \mathcal{L}_*$ in terms of the usual parameters μ, α, β, then we would be able to derive update rules for the parameters where we take gradients of the ELBO with respect to these parameters (e.g. $\nabla_\alpha \mathcal{L}$ instead of $\nabla_{m_\alpha} \mathcal{L}_*$). Then the next problem becomes the aforementioned swapping of the gradient of an expectation with the expectation of a gradient, in order to get gradients with respect to the weights instead of the distribution's parameters. We already mentioned a couple of ways to obtain this. An alternative approach is presented in [JMW14] where the goal is to find a non-linear function $B(x)$ such that for a function $f(x)$ and distribution $p(x|\theta)$ we have

$$\nabla_\theta \mathbb{E}_p[f(x)] = -\mathbb{E}_{p(x|\theta)}[B(x) \nabla_x f(x)] \tag{3.43}$$

where for exponential family distributions

$$B(x) = \frac{[\nabla_\theta \eta(\theta) \phi(x) - \nabla_\theta A(\theta)]}{[\nabla_x \log[h(x)] + \eta(\theta)^T \nabla_x \phi(x)]} \tag{3.44}$$

where $h(x)$ is the base measure, θ the set of mean parameters, η the set of natural parameters and $A(\theta)$ is the log-partition function [JMW14]. We derived the B function for the model discussed here. It can be found in appendix C.5.

The final approach we considered for deriving a hierarchical noisy optimizer is an approach we only touched upon briefly since it was presented late in the stage of this thesis by the author of [Kha18] during our aforementioned discussion with him. The approach essentially relies on two things. First of all, that the base distribution is a Normal distribution and secondly, that one chooses a conditional exponential family distribution. Because this approach was discussed so late in the progress, the derivation and discussion becomes rather fragmented and thus we omit this from the main part of the thesis, though it can be found in appendix C.

In conclusion, we did not manage to implement a hierarchical noisy optimizer though it was not for a lack of trying.

CHAPTER 4

Experiments & Benchmarks

To investigate the properties of Bayesian models versus deterministic models, we conduct a number of experiments. In order to execute the experiments in a controlled environment, where we can change the parameters of interest, all experiments were ran on simulated data.

We also benchmark our Bayesian model(s) on a classification and a regression dataset. Specifically, we consider the MNIST and Wine datasets (more details on the datasets follow in section 4.2.1). We note that the benchmarks serve the purpose of showing comparable performance between the various models tested here, and not for the purpose of replicating state of the art results. Thus, the results we obtain on these benchmarks can be further improved by greater hyperparameter optimization and further tuning of the models, such as implementing more Monte Carlo samples and in the regression cases tuning the variance of the data log-likelihood.

Lastly, we will also take a look at the sparsity inducing properties of hierarchical models by performing experiments and benchmarks on models using the student's t prior and the Horseshoe prior.

A note on the implementations All relevant code used in this thesis (except code related to implementations on Alipes Capital's model) can be found here: `https://bitbucket.org/RamiroCope/thesis_repo/src/master/`. Consult the readme file in the repo on code execution. All the noisy optimizers considered here have been implemented by ourselves though source code for Vprop have been made publicly available during the development of this thesis. The implementations of the hierarchical models are done in a BBVI framework. For all BBVI models we used a base BBB model which was readily available at `https://github.com/emtiyaz/vadam/tree/master/pytorch` and modified it to accomodate hierarchical models. Lastly, in order to stay true to the Adam optimizer we implement noisy Adam where we take the square root of the Fisher as mentioned in chapter 2.

4.1 Experiments on simulated data

Generally speaking, simulating data let us control parameters of interest in order to determine whether or not a model is behaving as desired. More specifically, for a Bayesian model where we assume the weights come from a distribution, we can control what the true distribution of the weights is and investigate if the model is able to learn the true distribution. This idea will be the basis for highlighting some of the strengths of the proposed optimizers (and Bayesian models in general) in the following experiments.

For the experiments, the simulated data, which we denote $\boldsymbol{y}^*$, was generated as follows:

$$\boldsymbol{y}^* = f(X\boldsymbol{w}^*) + \boldsymbol{\varepsilon} \tag{4.1}$$

Here $\boldsymbol{w}^*$ are the true weights, sampled from the true distribution, i.e. $\boldsymbol{w}^* \sim \mathcal{N}(0, \sigma^2\mathbf{I})$. Furthermore, the design matrix is generated by sampling $n \cdot m$ i.i.d random variables from a Normal distribution such that $X_{i,j} \sim \mathcal{N}(\mu, \sigma^2)$ where n is the number of observations and m the number of features. Lastly, we add noise to the targets, $\boldsymbol{\varepsilon} \sim \mathcal{N}(0, \sigma^2\mathbf{I})$. The function f determines the type of network we consider and can be viewed as the activation function in a neural network. If we have $f(x) = x$, the network reduces to a standard linear regression.

The simulation experiments were run as a comparison between a Bayesian model using noisy Adam and a deterministic model using the normal Adam optimizer.

4.1.1 Data Efficiency: Can we learn with less data?

Data efficiency refers to the ability to learn with less data, and in this section we will investigate how well our bayesian model is able to learn, dependent on how much data is available. Since the Bayesian model is resistant to overfitting, in that it is able to regularize the weights in the network, we would a priori expect the model to perform better (on test data) than the deterministic model when less data is available.

The experiment was constructed as follows: 100 weights were drawn from a true distribution, $\boldsymbol{w}^* \sim \mathcal{N}(0, \frac{1}{5^2})$ (we found this choice of variance to work best, especially for the cases where an activation function is considered). Next, the feature matrix X was constructed by drawing $X^{n \times 100} \sim \mathcal{N}(2, 4)$, where n (the total number of observations) increases for each training sequence. Lastly, we add noise $\boldsymbol{\varepsilon} \sim \mathcal{N}(0, 3)$ in order to generate our simulated targets. We considered a linear regression case ($f(x) = x$) and the case of a tangent hyperbolic activation function. We considered a data range of 100 to 4000 observations. Thus we would expect a very low train loss when only 100 observations are available (essentially a train loss of zero for a deterministic model) and a rather high test loss due to overfitting. Both train and test loss should converge to the optimal solution as we begin to increase the data. The loss function we consider is simply the mean squared error. In this case, the optimal solution is the variance of the noise ε. To realize this, note that if we perfectly reconstruct the true weights, i.e. for our learned weights $\hat{\boldsymbol{w}}$ we have $\hat{\boldsymbol{w}} = \boldsymbol{w}^*$, the mean squared error becomes

$$\frac{1}{N}(\boldsymbol{y}^* - \hat{\boldsymbol{y}})^T(\boldsymbol{y}^* - \hat{\boldsymbol{y}}) = \frac{1}{N}(X\boldsymbol{w}^* + \boldsymbol{\varepsilon} - X\hat{\boldsymbol{w}})^T(X\boldsymbol{w}^* + \boldsymbol{\varepsilon} - X\hat{\boldsymbol{w}}) = \frac{1}{N}\sum_i \varepsilon_i^2 \quad (4.2)$$

thus we see that the best we can do if we perfectly reconstruct the true weights is the sampled second moment which is the variance of the noise ε since $\mathbb{E}(\varepsilon) = 0$.

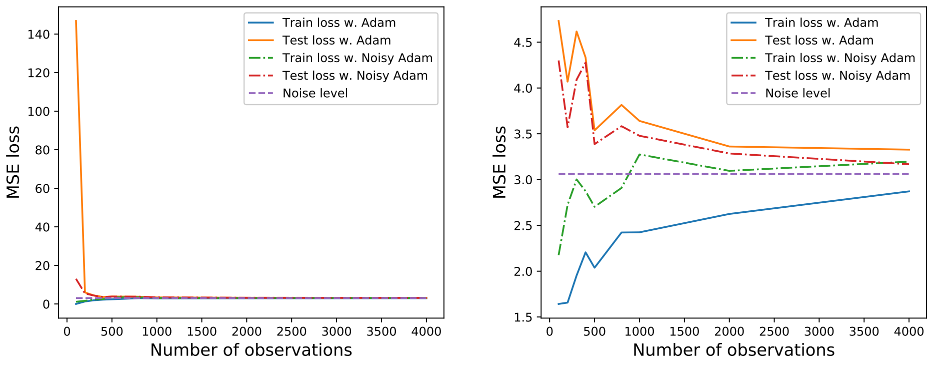


Figure 4.1: *The plot shows the test and train loss curves of deterministic and Bayesian neural networks under increasing data sizes. The BNN learns with less data in both plots.* ***Left:*** *Linear regression experiment.* ***Right:*** *Nonlinear experiment using the tangent hyperbolic activation function. Both models use a fixed weight size of a 100 and the starting point is set to 100 observations.*

Figure 4.1 shows the results of the experiments. For the linear regression experiment, it is clear that the deterministic model overfits severely when only 100 observations are available. Conversely, the Bayesian model is able to regularize some of the weights in such a way that it dramatically reduces the test loss when compared to the deterministic model. Not surprisingly the train loss reported for the deterministic model is essentially 0 (0.0004). We also see how the two models perform equally well as more data becomes available and both models converge to the noise level (variance of the noise) in tandem with the increasing number of observations (see figure 4.2 for a zoomed in view on the linear regression case). This is in line with the theory, since the prior should become less and less important as enough data becomes available. Lastly, we see the same behavior in the nonlinear experiment: the BNN is able to learn with less data.

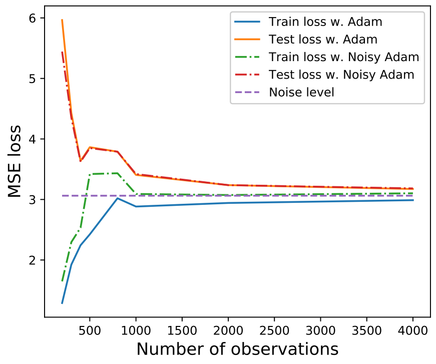


Figure 4.2: *A zoomed in view of the train and test loss curves for the deterministic and Bayesian model on the linear regression experiment. Note the convergence of both models towards the noise level.*

A central question remains. *How is the Bayesian model able to learn with less data?* To illustrate this point, consider an extreme example where only two data points are available. The deterministic model under a maximum likelihood estimate will fit a line connecting those two data points - a potential serious case of overfitting. The parameter fitting of the Bayesian model, however, does not depend solely on the likelihood term; we provide our belief as to how the true parameters are distributed via the prior. Thus, instead of fitting a straight line between the two data points, the bayesian model will fit a line that reflects a balance between the likelihood and the prior. As a result, the prior helps prevent overfitting when training data is scarce.

4.1.2 Model Complexity and Resistance to Overfitting

Increasing model complexity gives the model more flexibility to better fit data, however, it also runs the risk of not generalizing to unseen data, i.e. overfitting. Thus, model selection in deep learning, e.g. choosing the number of hidden units and layers that achieve the best performance, is of practical interest. We set up an experiment to evaluate how models behave as model complexity increases. The setup is the following: We sample a design matrix $X^{220\times200} \sim \mathcal{N}(0,1)$. We then create a one hidden layer neural network with 200 hidden units and use a sigmoid activation function. The weights of the neural network are drawn from a standard Gaussian. We then create the targets by doing a forward pass and adding standard Gaussian noise. Once the data pairs, $< X, Y >$ were simulated, we then fitted neural networks with increasing hidden units size to the data.

This experiment highlights one of the features of Bayesian deep learning models: auto-regulation. Figure 4.3 shows how the train and test loss remain relatively unchanged as the model complexity increases. The deterministic model on the other hand begins to overfit as the model complexity (number of hidden units) increases. This is expected of the deterministic model. A sufficiently powerful/flexible model without regularization will memorize/overfit a dataset and thus not generalize well to out of sample data. This is exactly what we see in the deterministic model. The test loss initially plummets, but with a very small amount of hidden units (10) it is flexible enough to overfit the training dataset - hence, the training and test losses diverge thereon after. The train and test losses for the BNN on the other hand do not diverge, but rather stay close together across the model complexity spectrum shown in the figure. Again, we attribute this behavior to the prior. A standard Gaussian prior is equivalent to L2 regularization, which penalizes large weight magnitudes and discourages complex models that do not explain the data. An interesting point to highlight is that when playing with the model complexity experiments as well as the others in this section, we observed that for the BNN not only that the train and test losses would follow each other closely, but also that in some instances the train loss would go above the test loss. I.e. the model would generalize better than what the train loss suggested. We could not explain this behavior and leave further analyses to understand this for future work.

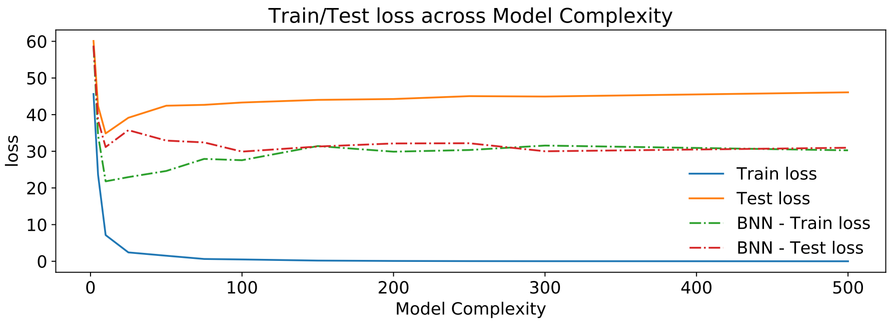


Figure 4.3: *The plot shows the test and train loss curves of deterministic and Bayesian neural networks under increasing levels of model complexity. Here model complexity refers to the number of hidden units in the neural network. The plot shows the deterministic network's tendency to overfit while the BNN is more resilient to it.*

4.1.3 Signal-to-Noise Ratio: Can we see through the noise?

Signal-to-Noise Ratio (SNR) tests are useful to understand a model's resilience to noise. This is an important feature in, for instance, financial applications such as stock price prediction, among others. The idea is that a noise-resilient model should be able to filter through the noise and detect the true underlying signal of the data.

We define the Signal-to-Noise ratio as the power p of the signal divided by the power p of the noise. Formally we have

$$\mathrm{SNR} = \frac{p_{\text{signal}}}{p_{\text{noise}}} = \frac{\frac{1}{N}\sum_i \sigma^2_{i,\text{signal}}}{\frac{1}{N}\sum_i \sigma^2_{i,\text{noise}}} = \frac{\sum_i \sigma^2_{i,\text{signal}}}{\sum_i \sigma^2_{i,\text{noise}}} \tag{4.3}$$

We construct two experiments to reflect the model's performance under different levels of SNR. The first experiment pertains to a linear model (linear regression) while the second to a nonlinear model using the tanh activation function. We simulate the targets in the same way as section 4.1.1, with a feature matrix of size 128×100 (i.e. 128 observations and 100 features). We consider a range of SNR from 1 to 10. That is, we iterate over training instances starting from equal amounts of signal and noise, to in the end 10 times more signal than noise. Figure 4.4 shows the results obtained. It is clear that the Bayesian model is better at capturing the true signal compared to the deterministic model in both linear and nonlinear scenarios. This can again be attributed to regularization of the network via the prior. Without regularization we would expect a sufficiently flexible model to fit the noise as well. The deterministic model, for instance, exhibits lower training losses yet higher test losses, which is diagnostic of learning the noise and thus not generalizing well to unseen data. It is worthy to note, however, that with enough observations, the deterministic network is able to perform comparable to the BNN, which again can be attributed to the prior becoming less and less important as we saw it in the data efficiency experiment.

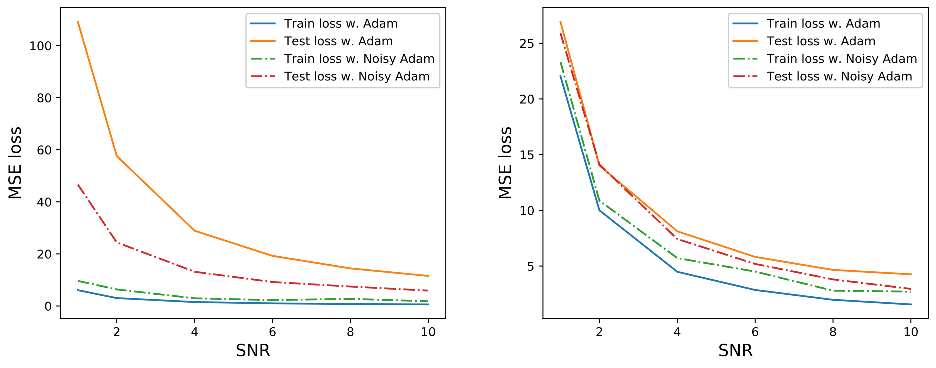


Figure 4.4: *The plot shows the test and train loss curves of deterministic and Bayesian neural networks under increasing levels of Signal-to-Noise ratio. Both models use a fixed weight size of 100.* ***Left:*** *Linear regression experiment: The BNN learns better under all SNR cases.* ***Right:*** *Nonlinear regression experiment using the tangent hyperbolic activation function: The BNN learns better under all SNR cases as well.*

4.1.4 Quantifying Uncertainty

One of the main motivations of Bayesian Deep Learning is to have a principled way of quantifying uncertainty. In cases when the model is presented with *out of distribution* data, e.g. an autonomous car driving in exceptional weather circumstances, we would like to know not only how the model extrapolates to unforeseen data, but also its confidence in those predictions. In this section we briefly showcase how we can obtain uncertainty estimates in regression and classification tasks. It is worthwhile to note that uncertainty can be broken down into two categories: *aleatoric* uncertainty and *epistemic* uncertainty. Aleatoric uncertainty stems from noisiness in the data. For instance, the labels in our classification task might not be entirely accurate, or the measurements for our regression task might include unknown levels of noise. Epistemic uncertainty, on the other hand, stems from uncertainty in the model's parameters and its structure. In this thesis, we focus on the epistemic uncertainty stemming from the model's parameters.

To showcase the uncertainty estimates in a regression task, we generated $< x, y >$ data pairs from the following function:

$$y = x + 0.3\sin(2\pi(x+\epsilon)) + 0.3\sin(4\pi(x+\epsilon)) + \epsilon \tag{4.4}$$

where $\epsilon \sim \mathcal{N}(0, 0.02)$. We fitted a BNN using Noisy Adam to the simulated data. Figure 4.5 shows the uncertainty estimates of the model's predictions.

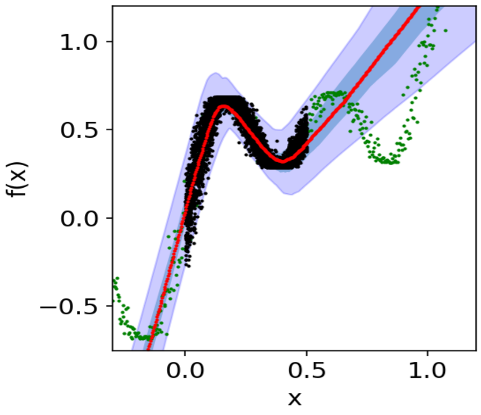


Figure 4.5: *Uncertainty in regression task learned with a BNN. The black and green dots are training and test points, respectively. The red line shows the median prediction while the blue and purple shades represent interquartile ranges.*

As can be seen, the model becomes increasingly more uncertain as the inputs $\boldsymbol{x}$ are further away from the training set. Given that the test data is outside the input training range, there are many possible extrapolations to the predictions. Thus, while the model is able to extrapolate to unforseen data, its uncertainty also gives us valuable information as to how confident the model is given particular $\boldsymbol{x}$ inputs. For the classification case, we illustrate the uncertainty output of a BNN trained with Noisy Adam on a simple univariate binary classification task. As Figure 4.6 shows, the model is quite certain in areas where the classes do not overlap. However, for $\boldsymbol{x}$ inputs in the range $0.4 - 0.6$ where the classes overlap, the model becomes increasingly uncertain. This is a desirable characteristic in many domains and in AI Safety in general. For instance, in domains where false positives have a high cost, such as in medical applications, we would like to know the uncertainty of the model when it is near the decision boundary. This information could then prompt medical professionals to do further studies and/or take the learning algorithm's prediction with caution.

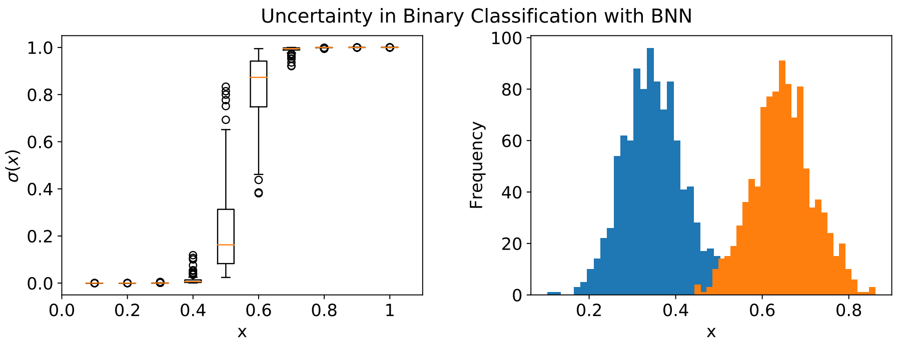


Figure 4.6: *Plots showcasing the uncertainty output of a BNN on a simulated single variable binary classification task.* ***Right:*** *The boxplots show the uncertainty of the predictions evaluated at different x points. The y-axis shows the output of the BNN through a sigmoid function for binary classification. Predictions near $x = 0.5$ show large uncertainty as it's where the classes overlap. Predictions away from $x = 0.5$ have minimal uncertainty.* ***Left:*** *The blue and orange histograms correspond to classes 1 and 2, respectively. They overlap around $x = 0.5$, making it difficult for the BNN to classify correctly, hence the larger prediction uncertainty in that region.*

For both classification and regression tasks, we obtain uncertainty estimates with the following procedure. Consider an optimized Bayesian model. For each new input $\boldsymbol{x}^*$, we estimate the predictive variance $\text{Var}[\boldsymbol{y}^*]$ (i.e. the model's prediction uncertainty) by running multiple forward passes through the model. However, instead of using the MAP estimate of the weights during the forward pass, we sample weights from the variational posterior, set the weights of the model to the sampled weights, and finally make a prediction based on the sampled weights, $\boldsymbol{w}_s$. The algorithm in table 4.1 below shows the procedure, where $\boldsymbol{y}^*$ is the prediction obtained using the MAP estimate of the weights.

```
Algorithm 5 Uncertainty Estimation
Require: $S$ : Number of samples to get from variational posterior
for each sample $s$ in $S$ do
    $\boldsymbol{w_s} \sim q(\boldsymbol{w})$
    $\boldsymbol{y_s} \leftarrow \boldsymbol{f(w_s, x^*)}$
end for
$\mathrm{Var}[\boldsymbol{y}^*] \approx \mathrm{Var}[\boldsymbol{y_1}, \boldsymbol{y_2}, ..., \boldsymbol{y_S}]$
```

Table 4.1: *Uncertainty estimation algorithm*

4.2 Benchmarks

In the following, we benchmark the different models (optimizers) on the Wine and MNIST datasets as previously mentioned. We will be comparing the different noisy optimizers against each other as well as against a deterministic network and a Bayes by Backprop (BBB) variant. Specifically, we consider models employing noisy Adam, noisy KFAC, Vprop, BBB and a deterministic model using Adam.

4.2.1 Data Description

MNIST The MNIST dataset probably needs no introduction, nevertheless we will give a short description here. The MNIST dataset is a collection of handwritten digits, ranging from 0 to 9. In total there is 60000 handwritten digits in the train dataset and 10000 in the test dataset. Each handwritten digit is stored as an image of size 28×28 pixels, with each pixel ranging from 0 to 255. When training and testing our models, any given image is flattened to a $28 \cdot 28 = 784$ long vector which we normalize by dividing each individual pixel value by 255. Thus, each individual pixel can be seen as an input feature to our models. The task becomes classification where our model should predict what kind of digit it receives. Thus we naturally employ the Cross-Entropy loss for this task.

Wine The Wine dataset is taken from the UCI (University of California Irvine) machine learning repository and contains various information about red wine. Specifically, the dataset contains 1599 observations and 11 features, with the quality of the red wine being the response variable ranging from 0 to 10 in score. To mention a few of the features, we have e.g. pH, alcohol, sulphates, various acidities etc. The task is regression, trying to predict the quality of the red wine. We use the mean squared error as loss function for this task. One could also have chosen the Gaussian Maximum Likelihood Estimator (MLE) in this case and estimated the variance of the MLE, however we choose not to do that in this case. Lastly, note that we normalized the data by subtracting the mean and dividing by the standard deviation where the mean and standard deviation is computed based on the training dataset.

Technical Note To have a fair comparison between the performances of the models in the benchmark datasets (MNIST, Wine, Alipes Data), we used Bayesian Optimization to tune the hyperparameters.

4.2.2 Regression

In this section we benchmark our different models on the Wine dataset (described above). In order to have comparable results between the different models, we for all models consider a simple feedforward neural network with a single hidden layer containing 50 hidden units. We optimized hyperparameters for all models using Bayesian optimization. For noisy Adam and noisy KFAC we optimized the KL weighting term and the prior variance. For Vprop and BBB we optimized the prior variance. Results are shown in table 4.2 and figure 4.7.

We see that the models employing noisy KFAC, BBB and Adam all converge more or less to the same level around 0.6 in test loss whilst the models with noisy Adam and Vprop perform a little worse. Surprisingly, it seems that as far as convergence goes, both noisy Adam and Vprop trail behind. This seems to go against what is claimed in the noisy natural gradient paper [Zha17]. Several possibilities exist for why our results on the Wine dataset contradict what is shown in the paper(s). As mentioned, we only normalize the data and randomly split the dataset into train and test sets with 90% used for training. They repeated the splitting process of the data (train/test splits) 20 times to reduce randomness. Where we used the mean squared error, they used the Gaussian MLE, where they optimized the variance of the MLE (this is probably one of the main reasons for the difference in reported convergence). Lastly, we only estimate the data log-likelihood with a single Monte Carlo sample. Increasing the number of samples could potentially increase the speed of convergence at the cost of increased runtime, however if the number of Monte Carlso samples was the problem, we would probably see it reflected in the BBB model as well.

From table 4.2 we conclude that the deterministic model performs best while for the models performing variational inference, the one using noisy KFAC performs best. The three remaining models exhibit similar performance. It is interesting to note the runtime of the different models, since faster runtime should be one of the key advantages of the noisy optimizers over say, a BBB solution. The runtime results confirm this, with the models employing noisy optimizers being only slightly slower than the deterministic model, while the BBB model is significantly slower.

	Adam	**noisy Adam**	**noisy KFAC**	**Vprop**	**BBB**
Test RMSE	$\mathbf{0.759 \pm 0.006}$	0.781 ± 0.009	0.767 ± 0.005	0.789 ± 0.004	0.78 ± 0.023
runtime	**0.88s**	1.03s	1.12s	1.00s	2.45s

Table 4.2: *Performance of the different models on the Wine dataset. Note that the Test RMSE reported is the averaged normalized test RMSE plus/minus the standard deviation based on 20 repeats.*

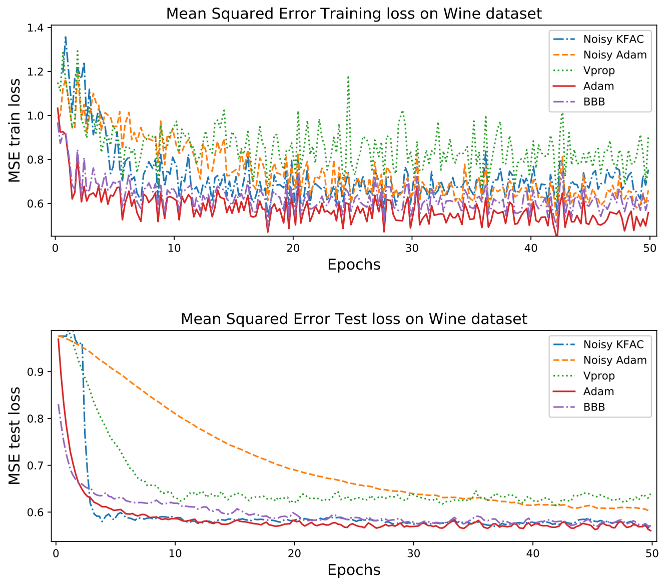


Figure 4.7: *Mean Squared Error for both train and test losses on the Wine dataset.*

As an illustration of the discussion on uncertainty estimates in section 4.1.4, we follow the algorithm in table 4.1 to obtain uncertainty estimates for our learned Bayesian model on the Wine dataset (the model considered here employs the noisy Adam optimizer). Figure 4.8 shows 30 predictions of wine quality along with their uncertainties as boxplots as well as the true targets as blue circles (the same plot with standard deviation bars instead of boxplots can be found in appendix B figure B.6). We see that while some targets fall within the predictions' uncertainty intervals, there are cases where they fall outside. Note for instance prediction #24 which is far from the actual target. However, the boxplot is also very wide demonstrating the uncertainty in this prediction. Conversely, for prediction #10 the boxplot is very narrow and the actual target is within this range, showing a more certain prediction of the wine quality in this case.

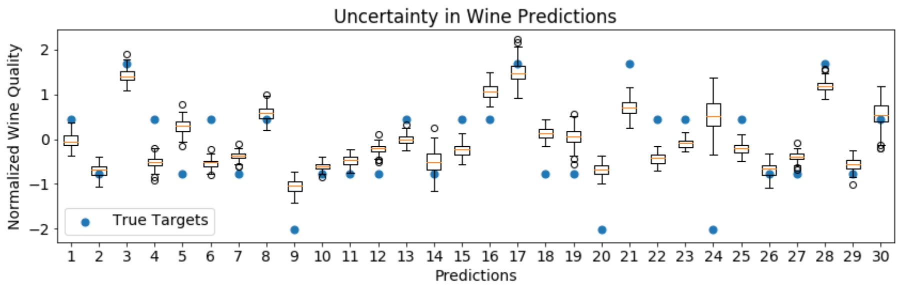


Figure 4.8: *Uncertainty of BNN in wine quality normalized predictions. The blue dots show the true targets (wine quality) while the boxplots show the median predictions as orange lines accompanied by their uncertainty as represented by the interquartile ranges.*

4.2.3 Classification

We now benchmark our models on the MNIST dataset. As for the regression benchmark, we for all models consider a simple feedforward neural network with a single hidden layer containing 100 hidden units. Hyperparameters were again optimized using Bayesian optimization. Results are shown in table 4.3 and figure 4.9 (also see figures B.7 and B.8 in appendix B for more detailed plots).

The story here is a bit different for MNIST compared to what we saw for the regression benchmark. From figure 4.9 it is evident that Adam is overfitting. The train loss is essentially zero, whilst the test loss is increasing in the last number of epochs. Conversely, all models performing variational inference have converged to their respective minimas. In this case, models employing noisy Adam, noisy KFAC and BBB converge more or less to the same level with noisy KFAC doing slightly better. Again, Vprop seems to be the worst performer, converging at a higher level than the rest. As far as convergence rates go, the story seems to be the same as for the regression benchmark. Noisy Adam and Vprop have the slowest convergence rates with noisy KFAC converging on par with BBB and Adam.

The test errors in 4.3 show that noisy KFAC performs best. With a test error of 1.82% it is performing as well as the small BBB model (two layers with 400 hidden units in each) as given in [Blu15]. Also note the performance of the deterministic model using Adam. Here the standard deviation of the predictions is higher than for the other models which can be explained by that fact that it overfits. Noisy Adam does more or less as well as BBB in this case while Vprop still falls behind. Finally note the runtime for the different models which tell the same story as the regression benchmark. The BBB model is significantly slower compared to the other models, whilst the models with noisy optimizers are slightly slower than the deterministic model.

	Adam	**noisy Adam**	**noisy KFAC**	**Vprop**	**BBB**
Test error %	2.33 ± 0.428	2.21 ± 0.110	$\mathbf{1.82 \pm 0.067}$	3.25 ± 0.110	2.12 ± 0.104
runtime	**5m49s**	6m36s	8m44s	6m38s	11m12s

Table 4.3: *Performance of the different models on the MNIST dataset. The test error has been averaged over 20 predictions and is reported together with the standard deviation. Runtime is given in minutes.*

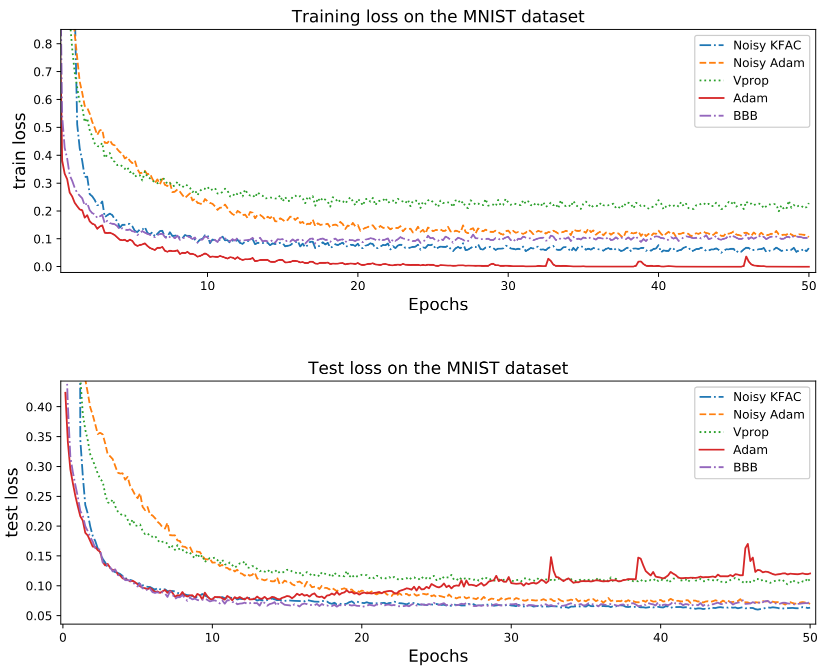


Figure 4.9: *Train and test losses for various models on the MNIST dataset.*

4.3 Experiments & Benchmarks on hierarchical models

We now take a look at BBVI models employing the student's t distribution and the Horseshoe as priors and the sparsity inducing properties they possess. For the implementation of the Horseshoe we initially tried simply running the source code for [GD17] which can be found here: `https://github.com/dtak/hs-bnn-public`.

However, the source code released is only for the simulation experiments presented in their paper. Thus we had to modify the code to train on MNIST. Furthermore, the code released is not written using `PyTorch` which complicated things. After modifying their code to run on MNIST we were not able to reproduce their reported results. In fact, the network did not learn at all. This led us to making our own implementation.

4.3.1 Flexibility of the student's t distribution

As we showed in chapter 3, a Normal distribution with a gamma distribution as a prior on the variance parameter, compounds to a student's t distribution. In this experiment we investigate the added flexibility of the student's t distribution in comparison with a Gaussian prior. Note that in this case the ELBO only changes in the way that we change both the prior and variational posterior distribution to the student's t distribution (i.e. both the prior and the variational posterior have the same form). As previously mentioned, this removes the hierarchy of the model since we still only consider distributions over the weights. Thus experiments and benchmarks using this configuration only serves to illustrate the slightly more sparsity inducing properties of changing ones prior to the student's t distribution.

We consider a simple linear regression case, where we simulate targets in the same way as done in the previous experiments, except the true weights no longer come from a pre-specified distribution. Instead we simply set 99% of the weights to zero, whilst setting the remaining 1% to large values in the range $[30, 60]$. Our hope is that the model using a student's t prior will have more weights concentrated around zero, as well as more larger weights when compared to the Gaussian prior. To that end we chose a prior with 1 degree of freedom and a prior variance of 1.

Figure 4.10 shows the resulting weights, obtained by taking the MAP estimate as usual (for a student's t distribution the MAP is still just the mean of the distribution). It is clear that the model with a student's t prior has more weights closer to zero. Furthermore, the model employing the Normal distribution prior is far from able to learn the large weights, whereas in the case of the student's t prior it is actually approaching the true large weights.

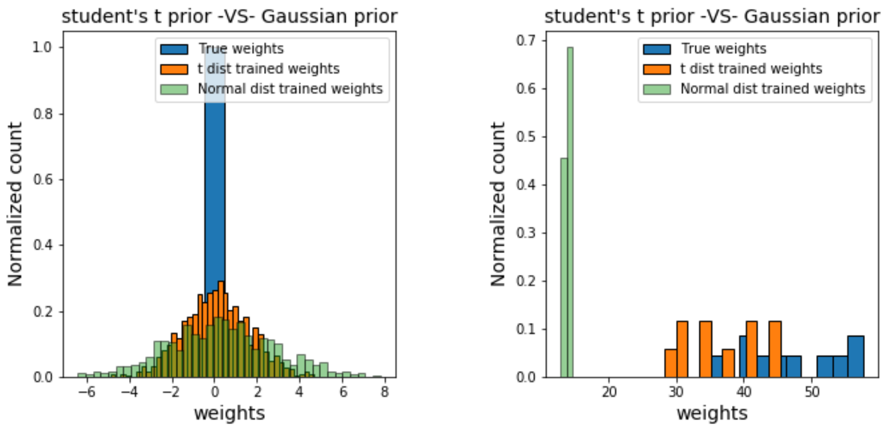


Figure 4.10: *Histograms showing the true weights and the learned weights for the two choices of priors.* ***Left:*** *Zoomed in view to show concentration around 0.* ***Right:*** *Zoomed in view to show larger weights learned.*

As table 4.4 shows, we counted how many weights are close to zero and close to the large true weights, i.e. how many weights are greater than 30, in order to get an idea of how well the two models regularize and learn the true weights. We see that a larger amount of weights are close to zero when considering the student's t prior. The story is the same when looking at the amount of weights greater than 30, where for a Normal distribution prior, the largest weights learned lies around 10 to 15. In conclusion, we have shown that by choosing a student's t distribution as our prior, we end up with a model which is more flexible in terms of regularizing weights towards zero whilst still being able to maintain large weights. This is useful in neural networks since the model becomes less sensitive to an increasing number of hidden units due to its regularizing capabilities.

	$\#\boldsymbol{w}$	$\#\lvert\boldsymbol{w}\rvert < 0.05$	$\#\lvert\boldsymbol{w}\rvert < 0.25$	$\#\boldsymbol{w} \geq 30$
true weights	1000	990	990	10
student's t	1000	20	134	9
Gaussian	1000	15	85	0

Table 4.4: *Count data of generated weights, student's t learned weights and Normal distribution learned weights for different threshold values.*

4.3.2 Group sparsity Horseshoe

To investigate the sparsity inducing properties of the Horseshoe, we consider a deep neural network (2 hidden layers with 200 neurons) where we simulate targets the same way as was done in section 4.1.2, except that we set the weights equal to zero for all neurons except the last neuron where weights were sampled uniformly in the range $[4, 6]$ (we do this for the two hidden layers while the output layer was sampled uniformly in the range $[-0.1, 0.1]$). Since our model employs group sparsity we would expect the model to turn all neurons off except for a single neuron which should try and capture the active neuron. It is worth mentioning however, that from our experience it is no easy feat to make the Horseshoe behave as expected, and one can spend considerable amounts of time on toying with hyperparameters. This is probably related to the fact that, unlike a Gaussian prior which has two tunable parameters to learn, the Horseshoe has 10. Thus, the search space for proper initialization of parameters explodes in comparison with the Gaussian. Likewise, searching for an optimum becomes more difficult and demanding since the search space now is that much larger.

Figure 4.11 shows a boxplot for the weights of the 50 last neurons in the first hidden layer of the model (see figure C.2 in appendix C for a boxplot of all 200 neurons). The boxplot clearly demonstrates both the general sparsity inducing property as well as the group sparsity property of the Horseshoe. Weights for all neurons except the last neuron (neuron #50) are closely centered around zero while the last neuron has moved in the direction of the true weights used when generating the simulated targets.

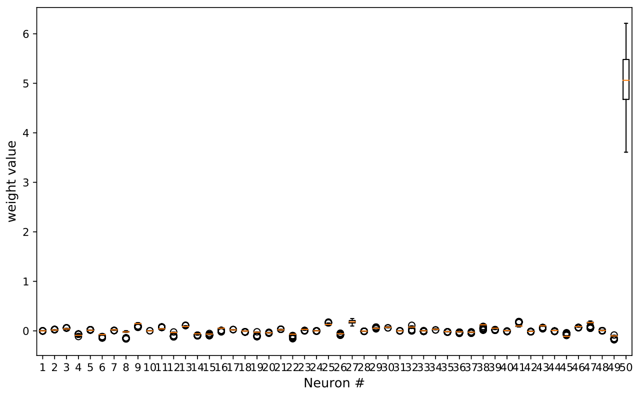


Figure 4.11: *Boxplot showing the weights for 50 of the 200 hidden units in a single layer in the simulation experiment. Note how the weights of all hidden neurons except one are closely centered around zero. The weights of the last neuron tries to capture the true weights which were sampled uniformly in that range.*

It is interesting to investigate exactly how close to zero the weights actually are for the model with the Horseshoe prior. Like we did for the student's t prior, we can count the number of weights which are below some tolerance level and compare with the Gaussian prior. Table 4.5 shows the results. There is a dramatic difference between the Horseshoe and the Gaussian prior in the number of weights below the different tolerance levels which underlines the sparsity inducing property of the Horseshoe. However, it is interesting to note that for the tolerance level of 0.0001 only a few of the total number of weights in the network are below this level. From a theoretical point of view, one should be able to improve on this number due to the aforementioned infinitely tall spike around zero for the Horseshoe. However, one should not forget that there is both a global and a local variance parameter at play here, and the interaction between the two. Perhaps we have chosen too large values for the weights in the active neuron, resulting in the Horseshoe "flattening" out too much, trying to reach the large weights. We believe that with further hyperparameter tuning/investigation, the sparsity of the Horseshoe can be improved in this scenario.

	$\#\boldsymbol{w}$	$\#\lvert\boldsymbol{w}\rvert < 0.01$	$\#\lvert\boldsymbol{w}\rvert < 0.0001$	$\#\boldsymbol{w} \geq 1$
true weights	40000	39800	39800	200
Horseshoe	40000	31512	720	200
Gaussian	40000	553	3	4295

Table 4.5: *Count data of generated weights, Horseshoe prior learned weights and Normal distribution learned weights for different threshold values.*

Lastly, it is worth taking a look at the weight distribution for the two models considered. Figure 4.12 shows histograms for the weights in the first layer of the respective models. For the Horshoe model we only plot weights which are below 0.1 (which are almost all the weights) in order to keep the histogram informative. We see how in the Gaussian case, weights go out as far as ± 6 and the concentration around zero is very "broad". For the Horseshoe we see a similar shape in the distribution of the weights, but note the range of weight values of ± 0.04. This explains the count data we presented in table 4.5 above. In conclusion, we have demonstrated the impressive sparsity inducing properties of the Horseshoe when compared to the classic Gaussian prior, however as we will see, this does not mean that we automatically get a model which outperforms simpler models on real datasets.

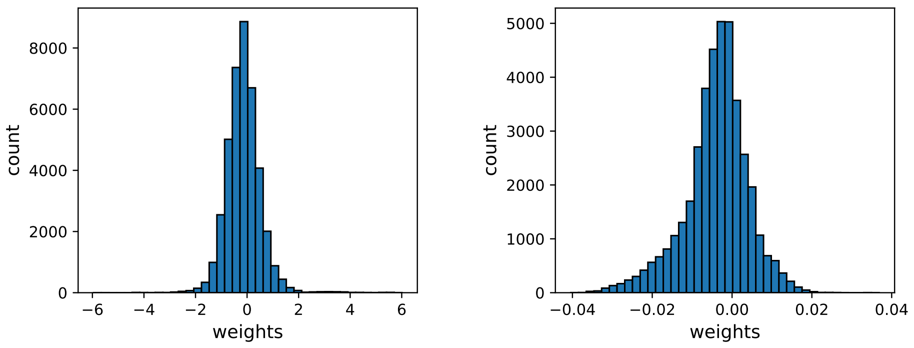


Figure 4.12: *Weight histograms for a BBB model using the Gaussian prior and a BBB model using the Horseshoe prior.* ***Left:*** *The Gaussian prior model.* ***Right:*** *The Horseshoe prior model where we have removed .*

4.3.3 Benchmarks on MNIST

In this section we briefly mention benchmarks of the models with student's t and Horseshoe priors on MNIST. We do not put too much emphasis on the benchmarks for these models, since we feel that it was more informative to show some of the properties of e.g. the Horseshoe rather than trying to obtain optimal performance.

4.3.3.1 student's t prior

For the model employing a student's t prior we consider a deep feedforward neural network (2 hidden layers with 200 units each) which we trained for 80 epochs. We obtained a test error of 1.75% which actually beats the performance achieved in [Blu15] for this given model configuration, though they use 400 instead of 200 units. Train and test plots are shown in figure 4.13 (see appendix B figure B.4 for KL divergence and ELBO plots). In comparison with the Gaussian prior we only do marginally better in this case (a 0.37% improvement). This is perhaps because we considered a prior student's t distribution with 6 degrees of freedom and a prior variance corresponding to the Gaussian prior case. Thus, in this case the student's t prior becomes a reasonable approximation to the Gaussian prior which potentially nullifies the benefits of using a student's t prior. The reason we chose the aforementioned prior configuration was because it, interestingly enough, produced the best results. Thus, perhaps a Gaussian prior simply works quite well on MNIST (which has been shown before) or we simply didn't find the right configuration for the student's t prior, though we did manage to outperform the model with the Gaussian prior.

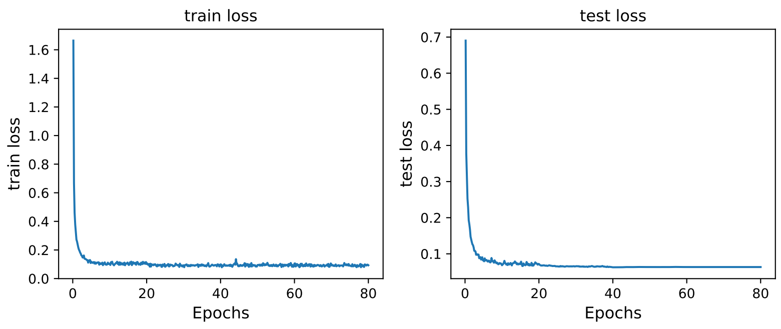


Figure 4.13: *Train and test losses on MNIST for a model employing the student's t prior.*

4.3.3.2 Horseshoe prior

For the model with a Horseshoe prior, the model configuration stays the same as for the student's t prior. We trained the model for 200 epochs. The results are shown in figure 4.14 (see figure B.5 in appendix C for KL divergence and ELBO plots). The model achieved a test error of 3.0%. The result is worse than most of the models discussed here. However, as previously mentioned, the number of parameters to be tuned is five times bigger than that of the usual BBB model. We spent what time we could afford on trying to find proper initialization values for the various parameters and in the end managed to find a configuration which at least provides comparable performance with the rest of our models.

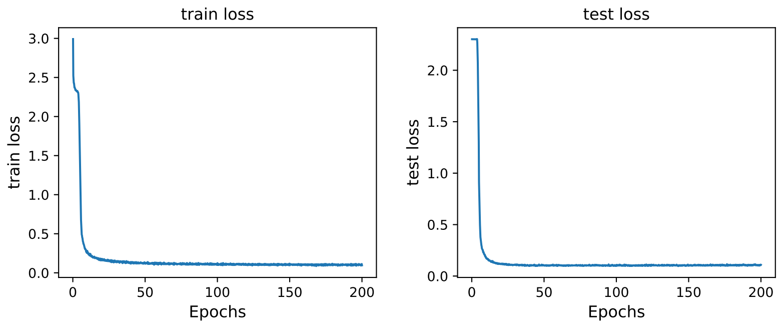


Figure 4.14: *Train and test losses on MNIST for a model employing the Horseshoe prior.*

CHAPTER 5

Financial Applications in collaboration with Alipes Capital

Not Available due to Non-Disclosure Agreement

CHAPTER 6 Future Work

A thesis only allows one to do so much within the time given. Thus there are a number of things we would have liked to explore further. Here we present some of them.

Hierarchical models on Alipes data The neural network developed by Alipes is more involved than e.g. creating a simple neural network in `PyTorch` which can be done in few lines of code. Their network is built with their own custom designed hidden layers and thus implementing a BBVI hiearchical model for Alipes poses its own challenges. As time was limited we did not get the chance to try out hierarchical models on Alipes' data. This would however be really interesting to investigate. Alipes have many correlated features in their dataset and one could potentially capture this and increase performance by considering a hierarchical model where the prior is more rich than when choosing a fully factorized prior. Consider for instance the model in (3.1)

$$\boldsymbol{w}|\boldsymbol{\lambda} \sim \mathcal{N}(\boldsymbol{w}|\mathbf{0}, \boldsymbol{\lambda}^{-1}), \quad \boldsymbol{\lambda} \sim \Gamma(\boldsymbol{\lambda}|\alpha, \beta)$$

if we instead of assuming that $\boldsymbol{\lambda}$ is a Gamma distributed vector now assume that it is Wishart distributed, we can begin to model dependencies between weights thus creating a more complex structured hierarchical model. We could then choose our Wishart distribution based on prior beliefs about the data, thus trying to for instance turn on or off groups of features or even model dependencies between them.

Experimenting with other priors The hierarchical models mentioned in this work are but an example of a wide range of different priors (each with their own properties) one can consider when doing hierarchical modelling. Thus, it would be interesting to investigate what properties other choices of a prior distribution might have. Consider the general setup for a weight w

$$w|\lambda \sim \mathcal{N}(0, \lambda^2), \quad \lambda \sim p(\lambda) \tag{6.1}$$

we consider the squared random variable λ here to stick with the notation used in e.g. [MGG09] and [GD17]. For this general setup, we can choose our hyperprior $p(\lambda)$ as we like. Besides the distributions mentioned in this thesis, we have come across distributions such as the Exponential distribution ($\lambda \sim \mathrm{Exp}(\lambda^2)$) and the improper log-uniform prior ($\lambda \sim p(\lambda) = \frac{1}{|\lambda|}$) which is known as the normal-Jeffreys prior. Setting the hyperprior to an exponential distribution would correspond to L1 regularization (LASSO) of the weights while the Jeffrey's prior, like the Horseshoe, corresponds to a Beta distribution $\mathrm{Be}(\alpha, \beta)$ where $\alpha = \beta \approx 0$. Thus it resembles the behavior of the Horseshoe. However, in contrast to the Horseshoe, the Jeffrey's prior can more or less only turn off or on the weights, where the Horseshoe still allows weights to reside in between these two options.

Minibatch noise analysis A way to potentially increase performance of the noisy optimizers, could be reduction of noise generated when doing minibatching. Currently, obtaining good performance with the noisy optimizers becomes a balancing act. On one side, we can obtain a decent approximation to the second order gradient by reducing the minibatch size as much as possible (recall gradient magnitude). On the other hand, reducing the minibatch size too much can slow learning for large datasets and also introduce excessive cumulated noise. By cumulated, we mean the inherent problem with these optimizers which generate noise by themselves through sampling which is added to the noise generated when doing minibatching. Thus future exploration and analysis of minibatch noise could be of considerable interest when using noisy optimizers.

Extending noisy KFAC Our implementation of noisy KFAC is quite limited in that we implemented a version which only supports two dimensional tensors. Furthermore, we only implemented a Matrix Variate Gaussian (MVG) version which assumes independence between the hidden layers in the network. In [Zha17] they implement a block tridiagonal approximation to the Fisher matrix which better captures dependencies between weights in adjacent layers. Thus, if time had permitted we would have implemented this as well, since the MVG version of noisy KFAC showed very promising results in our benchmarks, typically being the best performer. However, the implementation of a block tridiagonal Fisher approximation is a complicated matter and since our focus wasn't exclusively on noisy KFAC, we leave this for future implementations.

CHAPTER 7 Conclusion

In this work we have implemented variational inference algorithms for Bayesian Neural Networks which overcome some of the limitations of previous methods. We have shown that they exhibit beneficial properties such as robust uncertainty estimates, data-efficiency, overfitting resilience, and the ability to learn adequately in the presence of noise relative to deterministic models. Further, we have illustrated with benchmark datasets that they learn comparable to other variational inference methods yet with the added benefit of overcoming limitations such as ease of implementation, reduced gradient computations, and thus reduced fitting time. Lastly, we have described some of the challenges of extending these algorithms to hierarchical prior frameworks, and leave these extensions for future work.

APPENDIX A

Original optimizers

In this chapter we present the original optimizers as given in their respective papers.

A.1 Adam

```
**Algorithm** Adam
**Require:** $\alpha$: Stepsize
**Require:** $\beta_1$, $\beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
**Require:** $f(\theta)$: Stochastic objective function with parameters $\theta$
**Require:** $\theta_0$: Initial parameter vector
    $m_0 \leftarrow 0$ (Initialize 1st moment vector)
    $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
    $m_0 \leftarrow 0$ (Initialize 1st moment vector)
while $\theta_t$ not converged do
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
$\hat{m}_t \leftarrow m_t/(1 - \beta_1^t)$
$\hat{v}_t \leftarrow v_t/(1 - \beta_2^t)$
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$
end while
```

Table A.1: *Adam algorithm as presented in* [KB15]

A.2 RMSprop

Algorithm 1 RMSprop
```
$\boldsymbol{w} \leftarrow \boldsymbol{\mu}$
$\boldsymbol{g} \leftarrow \nabla_w f(\boldsymbol{w})$
$\boldsymbol{s} \leftarrow (1-\beta)\boldsymbol{s} + \beta(\boldsymbol{g}.*\boldsymbol{g})$
$\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} - \alpha(\boldsymbol{g}./\sqrt{\boldsymbol{s}+\delta})$
```

Table A.2: *RMSprop algorithm as presented in* [Hin]

A.3 KFAC

The algorithm for KFAC is rather extensive and presented in the original paper with references throughout the pseudo code to sections in the original paper. Thus we refer the reader to [MG15] if one would like to delve into the KFAC algorithm.

APPENDIX B
Optimizer statistics

Here we show the full set of statistics, namely train and test loss together with KL divergence and ELBO, obtained when training the various models, including the hierarchical models. We don't show KL divergence and ELBO for the model using the noisy KFAC optimizer. This is due to the fact that in order to compute the KL divergence, we need the covariance matrix (Fisher approximation) learned during training. To obtain the covariance matrix of the weights in a single layer we need to compute a Kronecker product:

$$\Sigma_l = S_l \otimes A_l \tag{B.1}$$

which for MNIST when using a hidden layer with 100 hidden units results in a covariance matrix for the first layer of size 78400×78400. This covariance matrix would have to be computed once per epoch or minibatch depending on the desired number of data points in the plot. This is hardly feasible to compute and thus we omit it in this case.

B.1 Noisy Adam

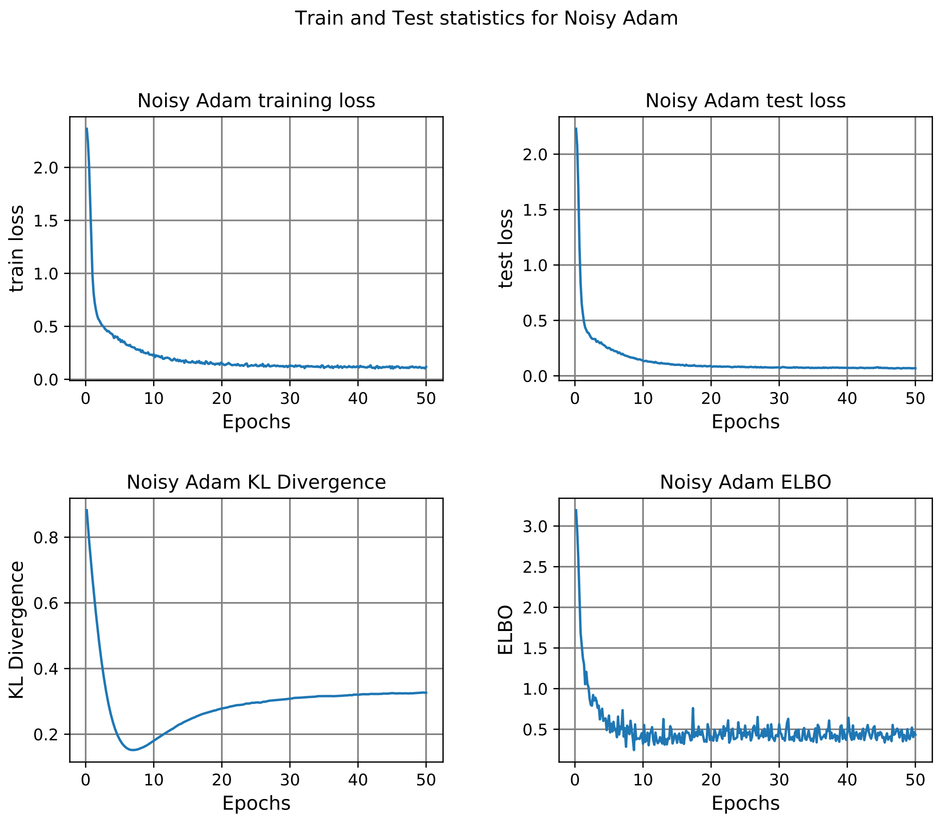


Figure B.1: *Per sample train and test loss together with KL divergence and ELBO using noisy Adam optimizer when trained on MNIST.*

B.2 Vprop

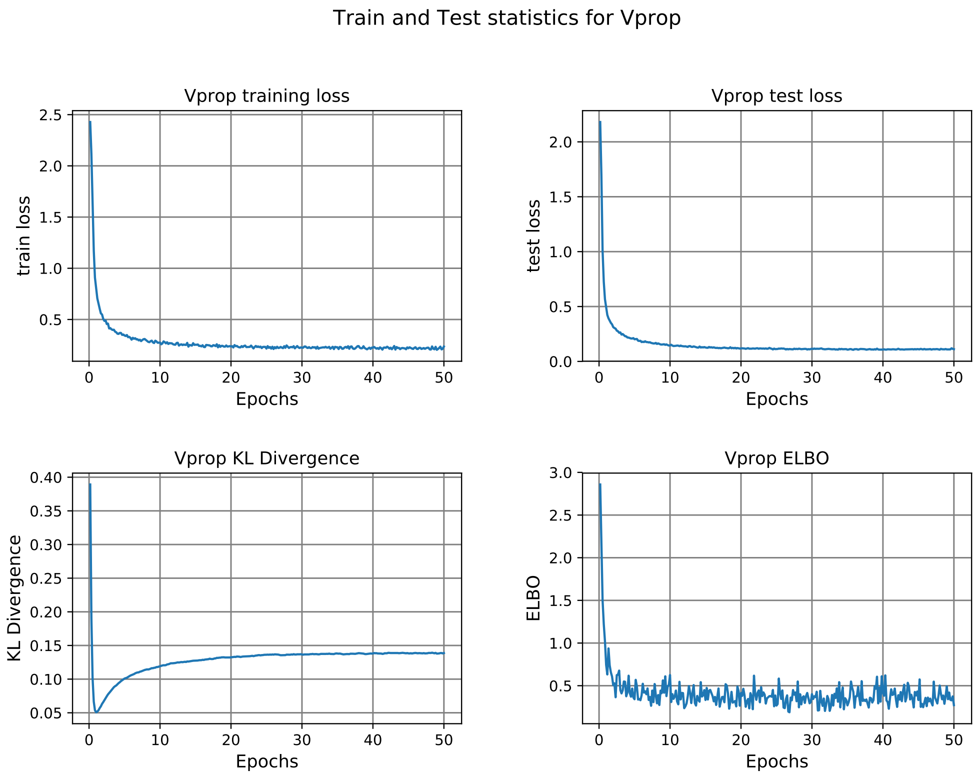


Figure B.2: *Per sample train and test loss together with KL divergence and ELBO using Vprop optimizer when trained on MNIST.*

B.3 BBB

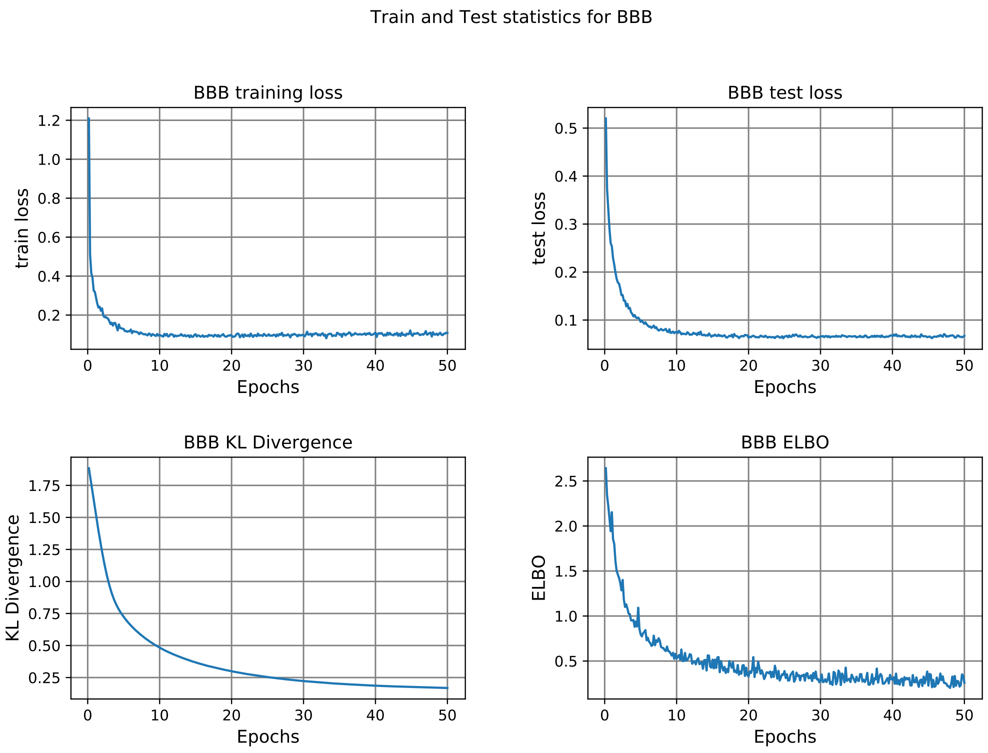


Figure B.3: *Per sample train and test loss together with KL divergence and ELBO using Bayes by backprop when trained on MNIST.*

B.4 student's t

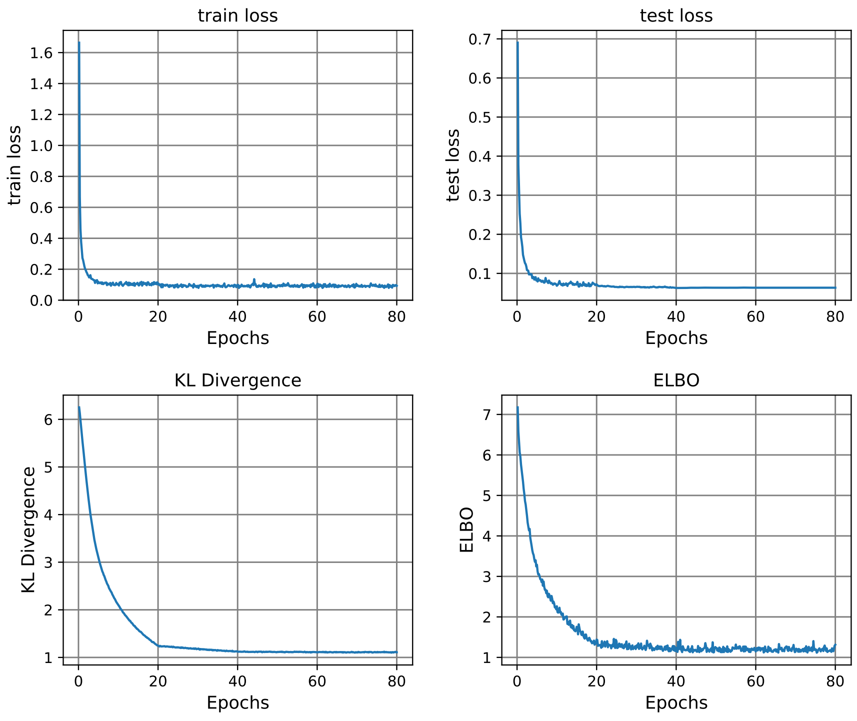


Figure B.4: *Per sample train and test loss together with KL divergence and ELBO using student's t prior when trained on MNIST.*

B.5 Horseshoe

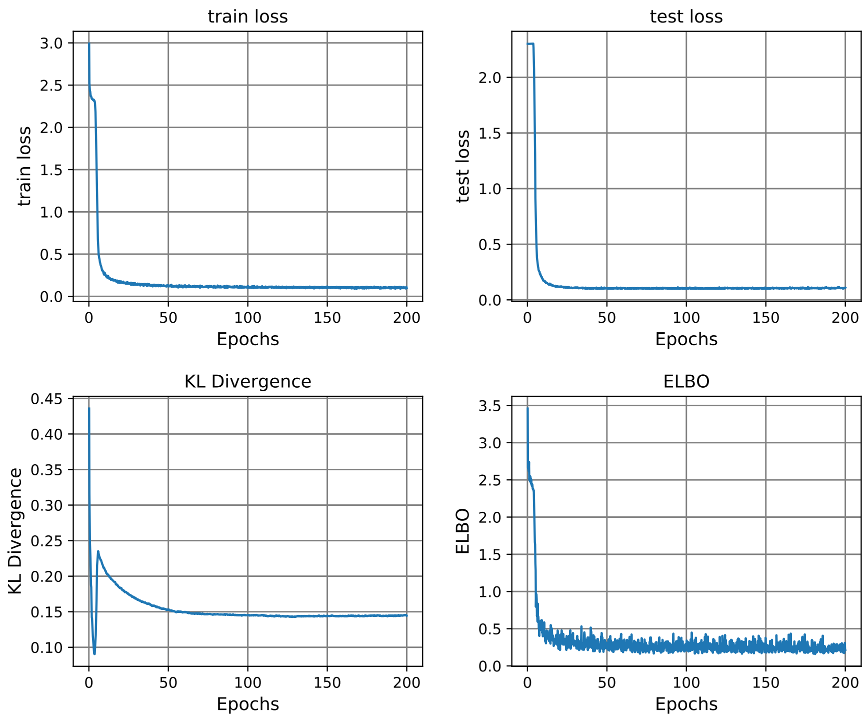


Figure B.5: *Per sample train and test loss together with KL divergence and ELBO using the Horseshoe prior when trained on MNIST.*

B.6 Wine uncertainty

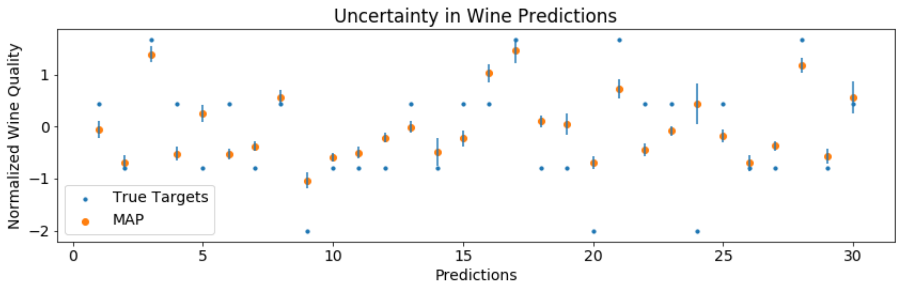
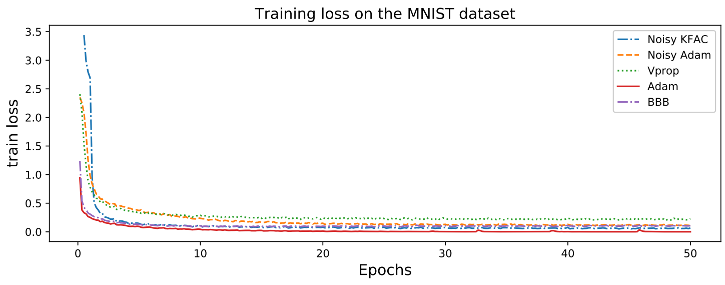


Figure B.6: *Model predictions with added uncertainty estimates for 30 predictions on the Wine dataset.*

B.7 Benchmarking all models on MNIST - Further plots



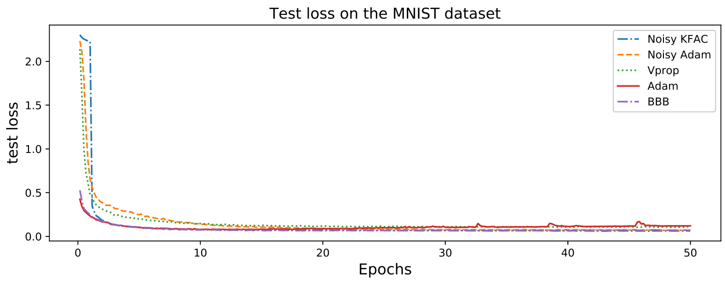


Figure B.7: *Complete plot of test and train losses for the different models when benchmarked on MNIST.*

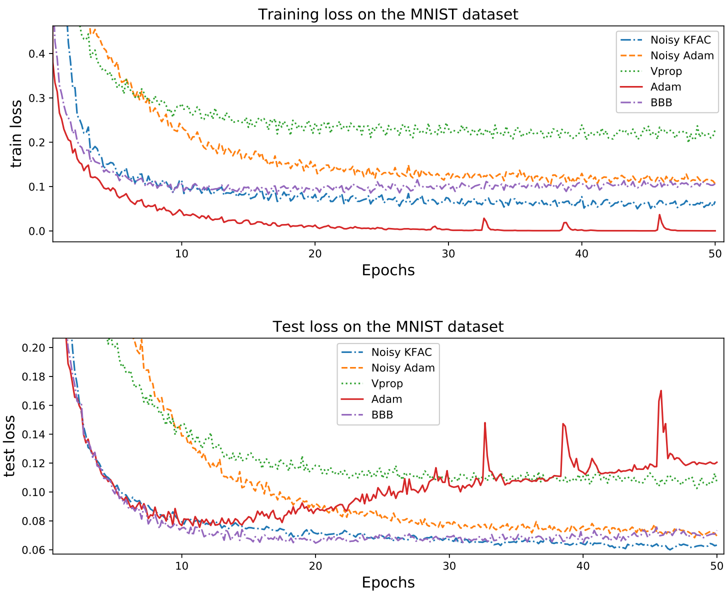


Figure B.8: *Zoomed in view of test and train losses for the different models when benchmarked on MNIST.*

APPENDIX C

Hierarchical models

Here we present various details on the hierarchical models mentioned in the thesis.

C.1 Cauchy densities

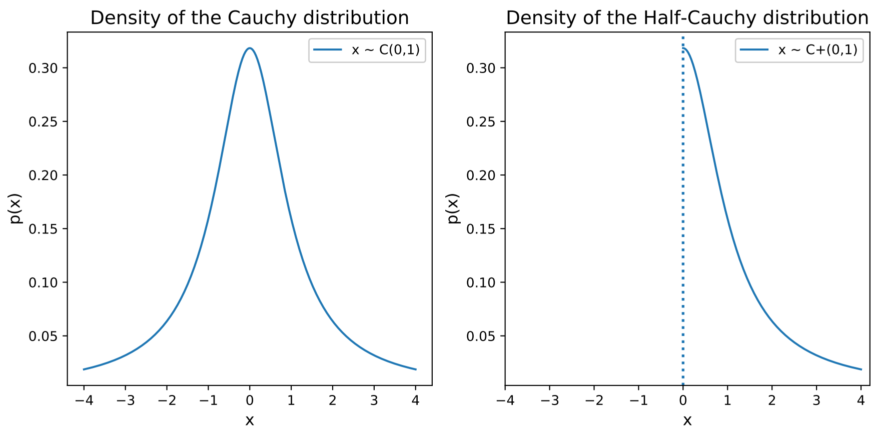


Figure C.1: ***Left:*** *Density of the Cauchy distribution $x \sim C(0,1)$.* ***Right:*** *Density of the Half-Cauchy distribution, which is simply the positive half of the Cauchy distribution.*

C.2 Forward pass for Horseshoe model

Let $\mathbf{H}$ be a minibatch of activations, $\mathbf{M}_w$ be the means of all the weights for a given layer and $\mathbf{\Sigma}_w$ be the variances of all the weights in a given layer. Then the forward pass of the Horseshoe model is presented in C.1 below.

Algorithm 1 Forward pass in layer l of Horseshoe

```
**Require:** H, $\mathbf{M}_w, \mathbf{\Sigma}_w$
$\hat{\boldsymbol{\epsilon}} \sim \mathcal{N}(0,1)$
$\mu_\tau = .5\mu_{\tau_a} + .5\mu_{\tau_b}$
$\sigma_\tau = \sqrt{.25\sigma^2_{\tau_a} + .25\sigma^2_{tau_b}}$
$\log \boldsymbol{\tau} = \mu_t au + \sigma_\tau \circ \hat{\epsilon}$
$\boldsymbol{\mu}_\lambda = .5\boldsymbol{\mu}_{\lambda_a} + .5\boldsymbol{\mu}_{\lambda_b} + \log \tau$
$\boldsymbol{\sigma}_\lambda = \sqrt{.25\boldsymbol{\sigma}^2_{\lambda_a} + .25\boldsymbol{\sigma}^2_{\lambda_b}}$
$\hat{\mathbf{E}} \sim \mathcal{N}(0,1)$
$\boldsymbol{\lambda} = \exp(\boldsymbol{\mu}_\lambda + \boldsymbol{\sigma}_\lambda \circ \hat{\mathbf{E}})$
$\hat{\mathbf{H}} = \mathbf{H} \circ \mathbf{Z}$
$\mathbf{M}_h = \hat{\mathbf{H}}\mathbf{M}_w$
$\mathbf{V}_h = \hat{\mathbf{H}}^2\mathbf{\Sigma}_w$
$\mathbf{E} \sim \mathcal{N}(0,1)$
return $\mathbf{M}_h + \sqrt{\mathbf{V}_h} \circ \mathbf{E}$
```

Table C.1: *Forward pass algorithm as presented in [LUW17]*

C.3 Horseshoe simulation experiment

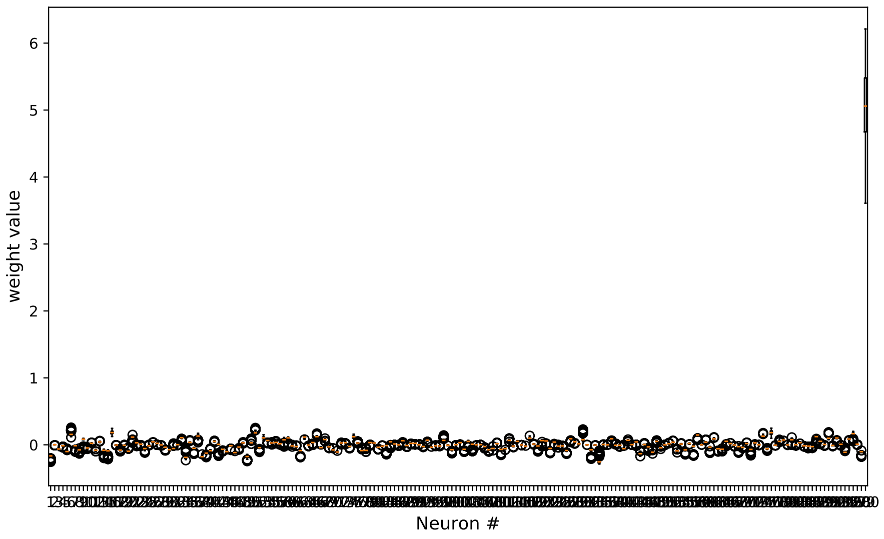


Figure C.2: *Full boxplot of all 200 hidden units in a single layer in the simulation experiment presented in section 4.3.2.*

C.4 Hierarchical noisy optimizer

As we mentioned in the main part of the thesis, an approach to implementing a hierarchical noisy optimizer is based on ensuring ones base distribution is a Normal distribution and that one chooses a general model setup which is a conditional exponential family distribution. In this section we will attempt to describe this approach as best we can.

Consider the general setup

$$w|\lambda \sim \mathcal{N}(0, \lambda^{-1}), \quad \lambda \sim p(\lambda) \tag{C.1}$$

In broad strokes, the main idea of this approach is that by obtaining a reparameterized sample from your variational posterior approximation to the hyperprior and using this sample in the base distribution, we still consider a Normal distribution when having to update our parameters. This should however be done in natural parameter space, before deriving the update rules for the mean and variance of the Normal distribution.

Formally, for some function $f(w)$ we can write

$$\mathbb{E}_{q(w|\lambda)q(\lambda)}(f(w)) \approx \mathbb{E}_{q(w|\lambda_*)}(f(w)) \tag{C.2}$$

where λ_* is a sample obtained through reparameterization using e.g Rejection Sampling Variational Inference [Nae+17]. Thus, we see that we only depend on the Normal distribution conditioned on the sample. By this reasoning, the updates for the mean and variance can be derived as in the usual way with the added twist that the ELBO changes slightly. As mentioned this should be derived in natural parameter space. To that end, consider the natural parameter $\eta = -\frac{1}{2}\Sigma^{-1}$ for the covariance Σ of a Normal distribution. The natural gradient update of η is given as

$$\eta_{t+1} = \eta_t + \beta_t \boldsymbol{F}^{-1}\nabla_\eta \mathcal{L}(\eta_t) \tag{C.3}$$

which according to theorem 2 in [Kha18] is equivalent to

$$m_{t+1} = \arg\min_m \langle m, -\nabla_m \mathcal{L}_*(m_t)\rangle + \frac{1}{\beta_t}\mathrm{D}_{\mathrm{KL}}(q_m(w)||q_{m_t}(w)) \tag{C.4}$$

where m is the expectation parameter, i.e. $m = \mathbb{E}_q(w^2)$. Now, because of the added hyperprior, the mirror descent step changes to

$$m_{t+1} = \left\langle \begin{matrix} m_w, \nabla_{m_w}\mathcal{L} \\ m_\lambda, \nabla_{m_\lambda}\mathcal{L} \end{matrix} \right\rangle - \frac{1}{\rho} D_{KL}(q_m(w)||q_{m_t}(w)) \tag{C.5}$$

where m_w is the expectation parameter for the variance of the Normal distribution and m_λ is the expectation parameters for the posterior approximation to the hyperprior. With this definition the Fisher matrix becomes a block diagonal

$$F^{-1} = \begin{bmatrix} [\nabla_{\eta_w}\mathcal{L}] & \\ & [\nabla_{\eta_\lambda}\mathcal{L}] \end{bmatrix}^{-1}$$

based on the parameters for both the Gaussian and the gamma distribution (though we are not sure how this truly is derived). The complication in this approach is how one obtains update rules for the parameters α and β of the gamma distribution. If the updates rules remain the same as for the Gaussian case, then one must somehow implicitly obtain update rules for the gamma distributions' parameters through the update of the variance of the Gaussian distribution.

C.5 Derivation of B function for scale mixture model

Here we derive the B function by following the definitions in [JMW14]. We consider the model given in (3.1) which for the parameters $\theta = \{\alpha, \beta, \mu\}$ has the following statistics

$$\boldsymbol{\eta}(\theta) = \left(\alpha - \frac{1}{2}, -\beta - \frac{\mu^2}{2}, \mu\right)^T \tag{C.6}$$

$$\phi(x) = (\log \lambda, \lambda, \lambda x)^T \tag{C.7}$$

$$A(\theta) = \log \Gamma(\alpha) - \alpha \log \beta \tag{C.8}$$

$$h(x) = \frac{1}{2\pi} \tag{C.9}$$

plugging these quantities into (3.44) yields

$$B(x|\alpha) = \frac{\log \lambda - \psi(\alpha) + \log \beta}{(\alpha - \frac{1}{2})\lambda^{-1} + \lambda(\mu - x)} \tag{C.10}$$

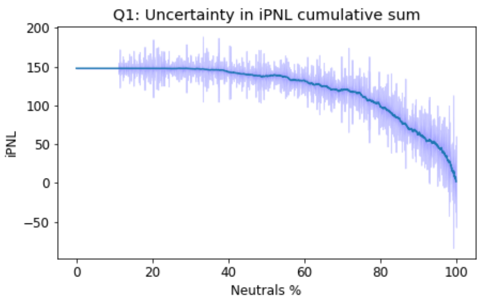
$$B(x|\beta) = \frac{-\lambda + \frac{\alpha}{\beta}}{(\alpha - \frac{1}{2})\lambda^{-1} + \lambda(\mu - x)} \tag{C.11}$$

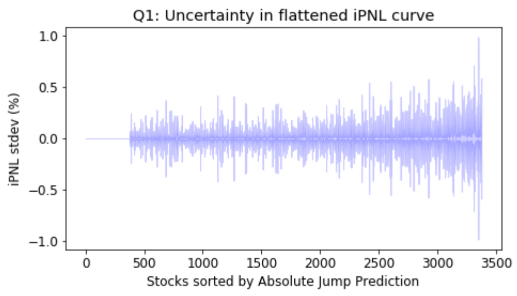
$$B(x|\mu) = \frac{\lambda x}{(\alpha - \frac{1}{2})\lambda^{-1} + \lambda(\mu - x)} \tag{C.12}$$

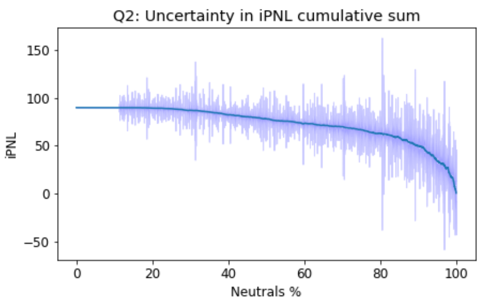
APPENDIX D

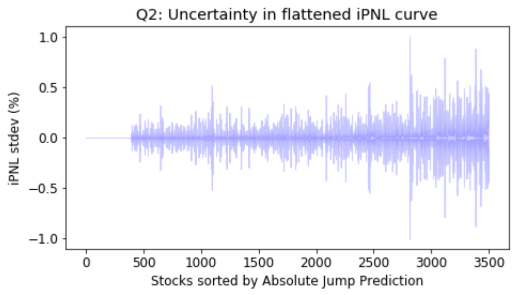
iPNL Uncertainty per quarter

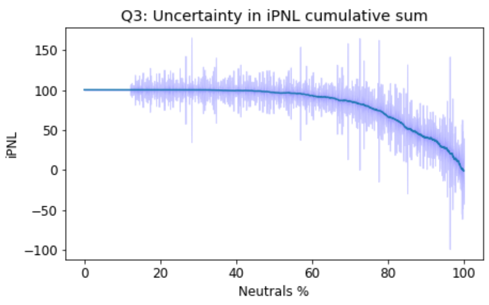
The plot shows the iPNL curve for Noisy Adam for the four different quarters of the test set. The uncertainty estimates shown on the left plots are exaggerated by 100% to be on the same y-axis scale as the cumulative iPNL. The right plots show the exact, non-exaggerated uncertainty iPNL estimates flattened to better compare uncertainty across stocks. Note that the stock ordering in both plots is identical. The iPNL in the right plots is not cumulative, thus its unit is in percent. The standard deviation shown is relative to that iPNL percent.

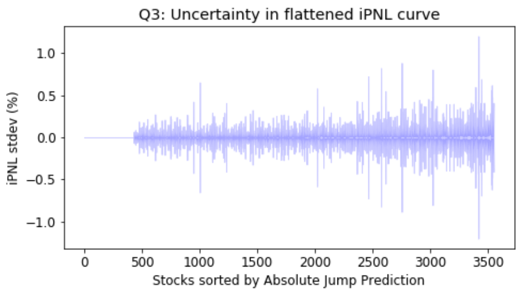


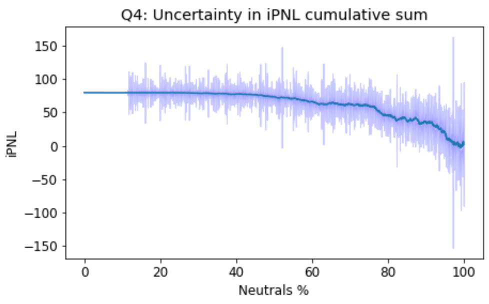


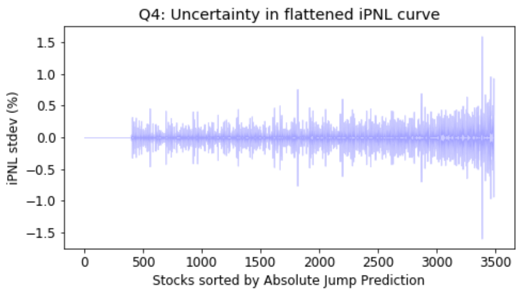





Q3: Uncertainty in iPNL cumulative sum
iPNL
Neutrals %


Q3: Uncertainty in flattened iPNL curve
iPNL stdev (%)
Stocks sorted by Absolute Jump Prediction


Q4: Uncertainty in iPNL cumulative sum
iPNL
Neutrals %


Q4: Uncertainty in flattened iPNL curve
iPNL stdev (%)
Stocks sorted by Absolute Jump Prediction

Bibliography

[18a] *Normal-gamma distribution.* 2018. URL: https://en.wikipedia.org/wiki/Normal-gamma_distribution (visited on August 4, 2018).

[18b] *Tutorial on Feedforward Neural Network — Part 1.* 2018. URL: https://medium.com/@akankshamalhotra24/tutorial-on-feedforward-neural-network-part-1-659eeff574c3 (visited on July 22, 2018).

[Ama16] Shun-ichi Amari. *Information Geometry and its Applications.* Applied Mathematical Sciences, Vol. 194. 2016. ISBN: 987-4-431-55977-1.

[Ama98] Shun-ichi Amari. "Natural Gradient Works Efficiently in Learning". In: (1998). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.452.7280&rep=rep1&type=pdf.

[Blu15] Charles Blundell et al. "Weight Uncertainty in Neural Networks". In: (2015). URL: https://arxiv.org/abs/1505.05424.

[Bot16] Léon Bottou et al. "Optimization Methods for Large-Scale Machine Learning". In: (2016). URL: https://arxiv.org/pdf/1606.04838.pdf.

[Dau+14] Yann N. Dauphin et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: (2014). URL: https://arxiv.org/pdf/1406.2572.pdf.

[FMM18] Michael Figurnov, Shakir Mohamed, and Andriy Mnih. "Implicit Reparameterization Gradients". In: (2018). URL: https://arxiv.org/pdf/1805.08498.pdf.

[GD17] Soumya Ghosh and Finale Doshi-Velez. "Model Selection in Bayesian Neural Networks via Horseshoe Priors". In: (2017). URL: https://arxiv.org/pdf/1705.10388.pdf.

[Gra11] Alex Graves. "Practical Variational Inference for Neural Networks". In: (2011). URL: https://papers.nips.cc/paper/4329-practical-variational-inference-for-neural-networks.pdf.

[Hin] Geoffrey Hinton et al. *Overview of mini-batch gradient descent.* URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[Hof17] Matthew D. Hoffman. “Learning deep latent Gaussian models with Markov chain MonteCarlo”. In: (2017). URL: http://proceedings.mlr.press/v70/hoffman17a/hoffman17a.pdf.

[JMW14] Danilo J. Rezende, Shakir Mohamed, and Daan Wierstra. “Stochastic Backpropagation and Approximate Inference in Deep Generative Models”. In: (2014). URL: https://arxiv.org/pdf/1401.4082.pdf.

[KB15] Diederik P. Kingma and Jimmy Lei Ba. “ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION”. In: (2015). URL: https://arxiv.org/pdf/1412.6980.pdf.

[Kha17] Mohammad Emtiyaz Khan et al. “Vprop: Variational inference using RMSprop”. In: (2017). URL: https://arxiv.org/abs/1712.01038.

[Kha18] Mohammad Emtiyaz Khan et al. “Fast and Scalable Bayesian Deep Learning by Weight-Perturbation in Adam”. In: (2018). URL: https://arxiv.org/pdf/1806.04854.pdf.

[KL51] S. Kullback and R. A. Leibler. “On information and sufficiency”. In: (1951). URL: https://projecteuclid.org/download/pdf_1/euclid.aoms/1177729694.

[Kno15] David A. Knowles. “Stochastic gradient variational Bayes for gamma approximating distributions”. In: (2015). URL: https://arxiv.org/pdf/1509.01631.pdf.

[LUW17] Christos Louizos, Karen Ullrich, and Max Welling. “Bayesian Compression for Deep Learning”. In: (2017). URL: https://arxiv.org/pdf/1705.08665.pdf.

[MG15] James Martens and Roger Grosse. “Optimizing Neural Networks with Kronecker-factored Approximate Curvature”. In: (2015). URL: https://arxiv.org/pdf/1503.05671.pdf.

[MGG09] Carlos M. Carvalho, Nicholas G. Polson, and James G. Scott. “Handling Sparsity via the Horseshoe”. In: (2009). URL: http://proceedings.mlr.press/v5/carvalho09a/carvalho09a.pdf.

[Nae+17] Christian A. Naesseth et al. “Reparameterization Gradients through Acceptance-Rejection Sampling Algorithms”. In: (2017). URL: https://arxiv.org/pdf/1610.05683.pdf.

[NOW14] Sarah E. Neville, John T. Ormerod, and M.P. Wand. “Mean field variational Bayes for continuous sparse signal shrinkage: Pitfalls and remedies”. In: Electronic Journal of Statistics, Vol. 8 (2014). ISSN: 1935-7524. URL: https://projecteuclid.org/download/pdfview_1/euclid.ejs/1407415580.

[OA09] Manfred Opper and Cédric Archambeau. “The Variational Gaussian Approximation Revisited”. In: (2009). URL: https://pdfs.semanticscholar.org/48dc/1de73230c3b1ff15d5aa20132fbdc31ad7d5.pdf.

[Rad95] Neal Radford M. “Bayesian Learning for Neural Networks”. In: (1995). URL: https://pdfs.semanticscholar.org/db86/9fa192a3222ae4f2d766674a378e470
pdf.

[TW15] Diederik Kingma Tim Salimans and Max Welling. “Markov chain monte carlo and variational inference: Bridging the gap”. In: (2015). URL: http://proceedings.mlr.press/v37/salimans15.pdf.

[Zha17] Guodong Zhang et al. “Nosiy Natural Gradient as Variational Inference”. In: (2017). URL: https://arxiv.org/abs/1712.02390.