University of Bergen
Department of Informatics

Gaussian Likelihoods in Bayesian Neural Networks

Author: Alvar Hønsi
Supervisors: Pekka Parviainen

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

October, 2023

Abstract

Bayesian neural networks (BNNs) offer a promising probabilistic take on neural networks, allowing uncertainty quantification in both model predictions and parameters. Being a relatively new and evolving field of research, many aspects of Bayesian neural networks still need to be better understood.

In this thesis, we explore the Gaussian likelihood function commonly used when modeling regression problems with Bayesian neural networks. Using variational inference, we train several Bayesian neural networks on synthetic datasets and investigate the Gaussian variance parameter (σ). We explore how it impacts the training process and shapes the resulting posterior distribution. We also explore an alternate approach where a prior distribution is placed on the variance parameter, and its value is inferred from the data.

While the data presented in this thesis is too limited to draw any definitive conclusions, we provide some interesting insights. We demonstrate that extreme values for σ can lead to tendencies of overfitting or underfitting BNNs. Additionally, inferring the variance parameter from the data can yield results on par with an "optimal" fixed parameterization of the likelihood function. We also showcase that misspecified Bayesian neural networks can produce overconfident uncertainty estimates and that inferring the variance parameter can help compensate for this limitation.

Acknowledgements

First of all, I would like to thank my supervisor, Pekka Parviainen, for his guidance and support throughout the project. Thank you for your patience, understanding, and your guidance through the scary world of Bayesian statistics. I would also like to thank my family and my girlfriend for their endless support and encouragement. Special thanks to my mom for proofreading the thesis and giving me great feedback. My heartfelt thanks to my fellow students and friends at Jafu for keeping me sane through the ups and downs of the last year. Lastly, I want to thank Eirik Rekve Thorsheim for always being helpful with administrative problems and for saving my degree on multiple occasions.

Alvar Hønsi
Monday 2nd October, 2023

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Since the introduction of the humble perceptron [1] in 1958, the artificial neural network (ANN) has been a topic of intense research. As a universal function approximator [2], the ANN has proven to be a powerful tool for solving a wide range of problems, especially in the last few decades, with large amounts of data and computing power becoming more readily available and the popularization of deep learning. They have been used to great effect in a wide range of fields, such as computer vision [3], which also encompasses object detection [4, 5] and facial recognition [6, 7], natural language processing [8, 9, 10], speech recognition systems [11] and many more. With the increasing popularity of ANNs, we have seen in recent years, it has also started to become more common to see them used in safety-critical applications, such as autonomous vehicles [12] and medical diagnosis software [13, 14, 15] . In these types of applications, where an erroneous decision made by a neural network can have severe consequences, it becomes increasingly important for the ANN to have explainability in its predictions and to be aware of its own uncertainty. This, however, is not a trivial task. ANNs, especially deep neural networks, are notoriously prone to overfitting and, when applied to supervised learning tasks [16] they are often incapable of correctly assessing the uncertainty of their predictions [17]. Introducing more data and regularization techniques such as dropout [16] can mitigate the overfitting problem. However, methods for accurately and truthfully assessing the uncertainty of a neural network is still an open problem.

In the last few years, there has been a resurging interest in the field of Bayesian neural networks, which will be the focus of this thesis. BNNs [18] are a probabilistic approach to ANNs, where the network weights are treated as random variables. BNNs offer a promising approach to the problem of creating neural networks with explainable

uncertainty estimates [19]. Much of the recent research has focused on improving the scalability and performance of approximate Bayesian inference methods suitable for BNNs [20, 17]. There have also been some deep dives into the impact of different priors on the resulting posterior distribution [21, 22, 23], which have provided much insight into the inner workings of BNNs. While there is a rich literature on the topic of BNNs and the related field of Bayesian deep learning [24], there is still much to be explored. In this thesis, we will look into the likelihood function of BNNs. We explore the Gaussian likelihood function used when modeling regression problems with BNNs. We explore how the choice of the variance parameter (σ) impacts the training process and shapes the resulting posterior distribution.

Our findings show the critical importance of well-specified likelihood functions, demonstrating that the choice of σ can have a significant impact on the resulting posterior distribution. We illustrate that overestimating or underestimating the noise in the likelihood function can lead to BNNs exhibiting overfitting or underfitting tendencies. We also explore the effectiveness of directly inferring a probability distribution over σ from the data and sampling its value from said distribution. Our findings show that inferred parameters for the likelihood function yield results on par with an "optimal" fixed parameterization of the likelihood function while removing the burden of specifying the parameter manually. Furthermore, we also show how inferring the likelihood parameter can help compensate for the often overconfident uncertainty estimates of misspecified BNNs.

1.1 Thesis structure

In chapter 2, we will give an introduction to the topics of Bayesian inference, artificial neural networks, and finally, bayesian neural networks. In chapter 3, we will discuss the motivation for this thesis. Chapter 4 details the experimental methodology used in this thesis and the implementation of the Bayesian neural network models. We will present the results of our experiments in chapter 5. Finally, we will discuss the results of our experiments in chapter 6, and present our conclusions in chapter 7. We will also discuss the limitations of our study and propose some ideas for future work.

Chapter 2

Background

In this chapter, we aim to cover most of the necessary background information needed to understand the concepts and methods used in this thesis. Section 2.1 introduces the basic concepts of artificial neural networks (ANNs), followed by section 2.2 which introduces the concept of Bayesian neural networks (BNNs). Finally sections 2.3 and 2.4 introduces the two main methods for performing approximate Bayesian inference in BNNs, namely Markov Chain Monte Carlo and variational inference.

2.1 Artificial Neural Networks (ANNs)

Artificial neural networks (ANNs) are a class of computational models inspired by the structure of biological nervous systems [25]. The basic building block of an ANN is a perceptron [1] (or neuron), a simple computational unit that combines a linear transformation z of the input with a non-linear activation function g. The linear transformation is defined as the weighted sum of the inputs x and a bias term b, while the activation function is applied to the result of the linear transformation. The output of the neuron is then defined as:

$$\begin{aligned} z &= (\sum_{i=1}^{n} w_i x_i) + b, \\ y &= g(z) \end{aligned} \tag{2.1}$$

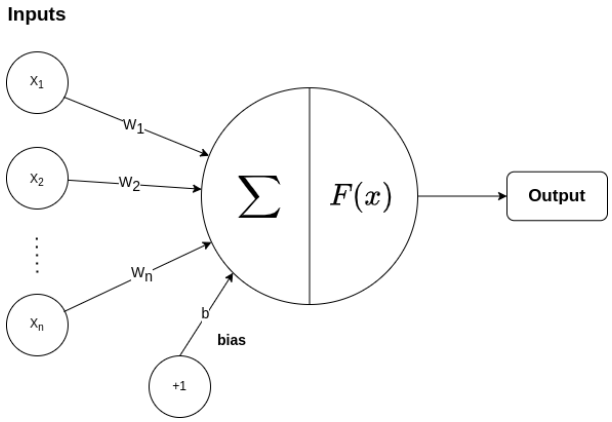


Figure 2.1: A single artificial neuron.

2.1.1 Feed Forward Neural Networks

Feedforward neural networks (FFNNs), also known as multilayer perceptrons (MLPs), are the most basic type of artificial neural network. A feedforward neural network aims to approximate some function f^*. To this end the network defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function approximation [26]. In this context, θ represents the weights and biases of the network, $\theta = \{w, b\}$. The term "feedforward" refers to the fact that the information flows through the network from the input x through any intermediate computations defined by f and finally to the output y. A feedforward neural network consists of multiple neurons organized into layers. The first layer is called the input layer, any intermittent layers are called hidden layers, and the final layer is called the output layer. The layers are fully connected, meaning that each neuron in a layer is connected to every neuron in the next layer. The output of each neuron in a layer is then passed as input to every neuron in the next layer. The output of the final layer is the output of the network. The weight parameters for a layer are usually represented as a matrix W where each row represents the weights for a single neuron in the layer and the bias parameters are then represented as a vector b. The output of a layer i can then be calculated as:

$$y_i = a(W_i^T y_{i-1} + b_i), \tag{2.2}$$

Where a is the activation function for the layer. The output of the previous layer

Figure 2.2 shows a feedforward neural network with two hidden layers.

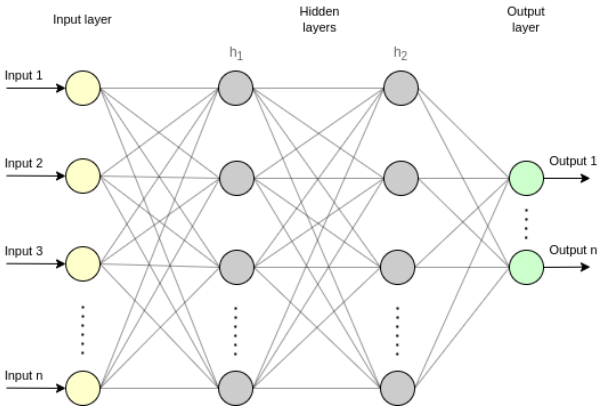


Figure 2.2: A feedforward neural network with two hidden layers.

2.1.2 Activation Functions

Activation functions are a critical part of the design of ANNs. The activation function defines how the weighted sum of the inputs is transformed into the output of the neuron.

Sigmoid

The sigmoid function is a commonly used non-linear activation function [27]. It is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.3}$$

The sigmoid function presents an S-shaped curve, which maps the input to a value between 0 and 1. Though the sigmoid function is a good choice for binary classification problems, it has some drawbacks. The gradient of the sigmoid function is small for values that are far from zero, which can lead to the problem of *vanishing gradients* during training.

ReLU

The rectified linear unit (ReLU) is another commonly used activation function and has been demonstrated to improve training in deep neural networks [28].

$$\text{ReLU}(x) = \max(0, x). \tag{2.4}$$

The ReLU function is simple to implement, computationally efficient, and non-linear, making it a good choice as an activation function. The ReLU function is also less prone to the problem of vanishing gradients since it has a constant gradient for values greater than zero.

2.1.3 Supervised Learning

Supervised learning is a type of machine learning where a machine learning model, such as a neural network, observes several samples of some input x and the corresponding output y and then learns to predict y from x [26]. This is done by defining some loss function $L(y, \hat{y})$, which measures the difference between the predicted output $\hat{y}$ and the true output y. The goal is to minimize the loss function by adjusting the model parameters θ. In other words, we want to find the optimal parameters θ^* which minimizes the loss function:

$$\theta^* = \arg\min_{\theta} L(y, \hat{y}). \tag{2.5}$$

2.1.4 Optimization

In order to find the optimal parameters θ^*, we need to optimize the existing parameters θ in order to minimize the loss function $L(y, \hat{y})$. The loss function describes how well the model fits the data, but we need to know how to tweak the parameters in order to lower the loss. By calculating the gradient of the loss function with respect to the weight parameters $\nabla L(\theta)$ we can determine how the loss changes when we change the parameter [26]. The gradient is a vector that points in the direction of the steepest ascent of the loss function. Therefore, the loss function can be minimized by moving in the opposite direction of the steepest ascent. To calculate the gradient we use the backpropagation algorithm [29] which employs the chain rule to calculate the gradient of the loss function with respect to each parameter in the network. Rumelhart et al. [29] also introduced a simple update rule for the parameters by accumulating the gradients over all data points and then updating the parameters according to the accumulated gradients $\nabla L(\theta)$ scaled by a learning rate η:

$$\theta_{t+1} \leftarrow \theta_t - \eta \nabla L(\theta_t). \tag{2.6}$$

In addition to this procedure, which is named gradient descent, there have been introduced other optimization algorithms such as Stochastic Gradient Descent (SGD) [30], which uses stochastic estimates of the gradient in order to increase efficiency and reduce computational cost. We also have the Adam optimizer [31], which maintains a learning rate for each parameter and adapts these learning rates during training.

2.2 Bayesian Neural Networks

Typically, when modeling neural networks, the weights are assumed to have a hidden true value, and the data is assumed to be random variables [19]. Through gradient optimization, we attempt to find the optimal point estimate of the weights, which is the most likely value of the weights given the data. This presents the standard frequentist approach to machine learning. However, from the view of Bayesian statistics, it makes more sense to treat the weight parameters as random variables, as they are unknown. We then want to learn the posterior distribution of the weights based on the information in the seen data. A neural network that is trained in this way, using Bayesian inference, is called a Bayesian neural network (BNN) [24].

During the learning process of a BNN, the unknown model parameters θ are hidden (or latent) variables, meaning that their true distributions are unknown. Bayes Theorem then allows us to represent a distribution over the weights θ given the observed data D, which results in the posterior distribution $p(\theta|D)$.

The joint distribution of the data and the weights can be expressed as $p(D, \theta)$, and is defined by our prior beliefs about the weights $p(\theta)$ as well as the choice of model and likelihood $p(D|\theta)$. The likelihood $p(D|\theta)$ can be viewed as the likelihood of generating the data D given the weights θ. If we break the data into a set of corresponding inputs and outputs $D = (x, y)$, then the likelihood can be expressed as $p(D|\theta) = p(y|x, \theta)$ - the probabilistic model by which the inputs generate the outputs given some parameter θ.

For simplification, it is often assumed that all samples from D are independent and identically distributed (i.i.d.), which allows us to express the joint distribution as:

$$p(D|\theta) = \prod_{i=1}^{N} p(y_i|x_i, \theta). \tag{2.7}$$

In the case of BNNs, the likelihood will be defined by the choice of neural network architecture and loss function. Given a regression task with a mean squared loss and known Gaussian noise, we can model the likelihood as a Gaussian distribution with the mean specified by the neural network output and the variance parameter σ, which is treated as a hyperparameter:

$$y \sim \mathcal{N}(f(x, \theta), \sigma^2), \tag{2.8}$$

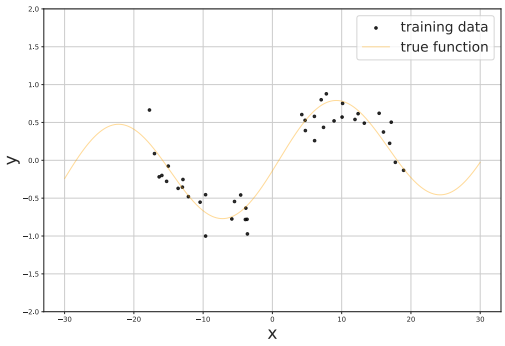
The Bayesian paradigm requires us to specify a prior distribution over the weights $p(\theta)$. The prior distribution encodes our beliefs about the weights before seeing the data. The most common choice for prior for the weights of a BNN is to use a simple isotropic Gaussian distribution [21], often due to its mathematical convenience. In the case of a zero-centered Gaussian with small variance, the prior will encode a bias for smaller weights centered around zero [32] and promotes sparsity in the model, which is a desirable property in artificial neural networks [33]. However, the Gaussian prior is by no means the optimal choice for all problems, as they are shown to be often misspecified, which can lead to negative consequences during inference [21]. As with other types of hyperparameters, the choice of prior distribution is often problem-dependent and should be chosen with care.

After specifying the prior distribution and the likelihood, we can use Bayes Theorem to calculate the posterior distribution of the weights given the data:

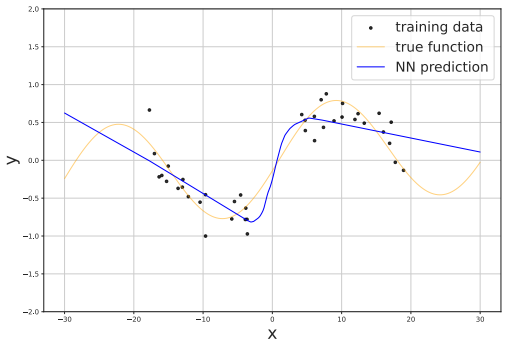
$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)} = \frac{p(D|\theta)p(\theta)}{\int p(D|\theta)p(\theta)d\theta}. \tag{2.9}$$

The Bayesian posterior over the parameters of a complex model such as a deep neural network is an extremely high dimensional and non-convex probability distribution [34]. This makes the posterior intractable and impossible to calculate in closed form, which we can already observe from the evidence (or marginal likelihood) $p(D) = \int p(D|\theta)p(\theta)d\theta$, requiring us to integrate over the entire parameter space. Because of the intractability of the posterior, we must instead use approximate inference techniques to approximate the posterior distribution. There are several useful approximate inference techniques that

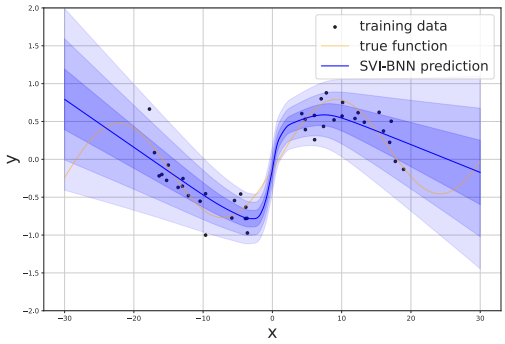
can be used to approximate an intractable posterior, but the most applicable ones for BNNs are Markov Chain Monte Carlo and variational inference [24]. These methods are discussed in detail in sections 2.3 and 2.4, respectively.



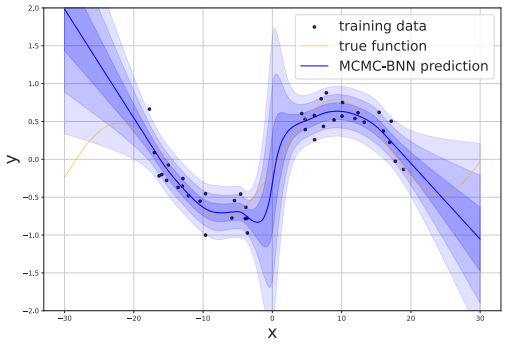
(a) Dataset



(b) NN



(c) SVI BNN



(d) MCMC BNN

Figure 2.3: Visualizations of predictions on the same dataset using different models. All models have three layers of 64 neurons each. Figure 2.3b shows the predictions of a standard neural network. Figure 2.3c shows the predictions of a Bayesian neural network trained using stochastic variational inference. Figure 2.3d shows the predictions of a Bayesian neural network trained using Markov Chain Monte Carlo. The shaded areas represent a $3 * std$ confidence interval around the mean prediction.

The above methods result in an approximate posterior distribution $p(\theta|D)$ whose variance can be seen as a measure of confidence in the model parameters. When performing predictions, we are interested in formulating a predictive distribution over the target variable y_i given the input x_i and the training data D. This can be done by marginalizing the model parameters θ:

$$p(y_i|x_i, D) = \int p(y_i|x_i, \theta)p(\theta|D)d\theta \tag{2.10}$$

Equation 2.10 represents a Bayesian Model Average (BMA) [35] where the predictive distribution is a weighted average of the predictive distributions of the individual models represented by the posterior distribution $p(\theta|D)$. This predictive distribution is in practice sampled indirectly [24] by sampling from the posterior distribution $p(\theta|D)$ and then using the sampled weights to make predictions. Algorithm 1 shows a general inference procedure for a Bayesian neural network.

Algorithm 1 Inference procedure in a Bayesian Neural Network [24]

Define $p(\theta|D) = \frac{p(D|\theta)p(\theta)}{\int p(D|\theta)p(\theta)d\theta}$;
for i = 0 to N **do**
    Sample $\theta_i \sim p(\theta|D)$;
    $y_i = f(x, \theta_i)$;
end for
return $Y = \{y_i | i \in [0, N)\}, \quad \Theta = \{\theta_i | i \in [0, N)\}$

Y is the set of predictions, and Θ is the set of sampled weights.
When performing predictions, these sets are usually aggregated to summarize the uncertainty of the model and to obtain an estimate $\hat{y}$ for the output y. For regression tasks, we can summarize the predictions by model averaging:

$$\hat{y} = \frac{1}{|\Theta|} \sum_{\theta_i \in \Theta} f(x, \theta_i) \tag{2.11}$$

2.3 Inference with Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) is a class of algorithms (or methods) for sampling from a probability distribution [36]. They have a wide range of applications but are especially useful in Bayesian Statistics and other fields where one needs to sample from a complex probability distribution. Sampling-based methods such as MCMC build on the idea of Monte Carlo integration [37], where we make us of the fact that the predictive distribution $p(y|x, D)$ can be expressed as an expectation over the posterior distribution $p(\theta|D)$:

$$p(y|x, D) = \int p(y|x, \theta)p(\theta|D)d\theta = \mathbb{E}_{p(\theta|D)}[p(y|x, \theta)] \tag{2.12}$$

which can then be approximated by sampling from the posterior distribution $p(\theta|D)$ and taking the empirical mean of the samples:

$$\mathbb{E}_{p(\theta|D)}[p(y|x,\theta)] \approx \frac{1}{N}\sum_{i=1}^{N} p(y|x,\theta_i), \quad \theta_i \sim p(\theta|D) \tag{2.13}$$

However, obtaining direct samples from an unknown high-dimensional and complex posterior distribution is a challenging task. In order to obtain these samples, MCMC methods make use of Markov chains [38]. Ergodic Markov chains have the property that they will eventually converge to a stationary distribution from any initial state. Using this property, MCMC methods construct a Markov chain with a stationary distribution equal to the posterior distribution $p(\theta|D)$. If the Markov chain is simulated for a sufficient number of steps, the samples will eventually converge to the stationary distribution. Samples from the chain can then be used to approximate the posterior distribution $p(\theta|D)$. MCMC methods have a significant advantage in that convergence is guaranteed as long as the chain is simulated for a sufficient number of steps. However, depending on the complexity of the posterior distribution and the starting state, the chain might take a very long time to converge. In addition to this, determining when the chain has converged can be challenging. In addition, early samples from the chain are often autocorrelated and, therefore, not representative of the target distribution. Because of this, MCMC methods usually require a burn-in period where the chain is allowed to converge to the stationary distribution before sampling [38].

2.3.1 Metropolis Hastings

The Metropolis-Hastings algorithm [39, 40] is a versatile and relatively simple MCMC algorithm which can be used to sample from any general probability distribution. The algorithm constructs a Markov chain with a stationary distribution proportional to a target distribution by sampling each new state from a proposal distribution and then accepting or rejecting the proposal based on a probability ratio. Given a target distribution p, a proposal distribution q and a current state θ_t, the a proposal state θ^* is sampled from the proposal distribution $q(\theta^*|\theta_t)$ and then accepted according to a transition probability α defined as:

$$\alpha(\theta^*|\theta_t) = \min\left\{1, \frac{p(\theta^*)q(\theta_t|\theta^*)}{p(\theta_t)q(\theta^*|\theta_t)}\right\} \tag{2.14}$$

If the proposal state θ^* is accepted, it gets added to the Markov chain and used for the next iteration; otherwise, it is discarded, and the current state θ_t is reused for the next iteration. As the algorithm accumulates more samples, the Markov chain will start converging towards the target distribution p. The Metropolis-Hastings algorithm is summarized in Algorithm 2.

Algorithm 2 Metropolis-Hastings algorithm [36]

Initialize θ_0.
for $t = 1$ *to* N **do**
  Sample $\theta^* \sim q(\theta^*|\theta_t)$.
  $\alpha = \min\left\{1, \frac{p(\theta^*)q(\theta_t|\theta^*)}{p(\theta_t)q(\theta^*|\theta_t)}\right\}$
  Sample $u \sim \mathcal{U}(0, 1)$.
  if $u < \alpha$ **then**
    $\theta_{t+1} \leftarrow \theta^*$
  else
    $\theta_{t+1} \leftarrow \theta_t$
  end if
end for
return $\theta_1, \theta_2, \ldots, \theta_N$

When it comes to the proposal distribution q, there are no strict requirements. A simple choice could be to use a Gaussian random walk proposal distribution centered around the current state θ_t [18], where the standard deviation would be chosen so that the acceptance probability remains reasonably high. However, Neal [18] also brings to attention some of the problems with this approach. Depending on the properties of the target distribution p and the proposal distribution, the Metropolis algorithm does not always produce an ergodic Markov chain, which means that the Markov chain will not visit all possible states with non-zero probability. Another problem with this configuration is that for larger high-dimensional distributions, such as Bayesian neural networks, big changes can quickly lead to regions of low probability in the sample space. Because of this, the standard deviation of the proposal distribution must often be set to a minimal value in order to maintain a reasonable acceptance probability. This again leads to highly correlated samples since many steps must be taken to reach distant points in the distribution. Due to these problems, which are further worsened by the random walk nature of the proposal distribution, the Metropolis algorithm is often very slow to converge for more complex distributions.

2.3.2 Advanced MCMC

Seeing as the Metropolis-Hastings algorithm is often very slow to converge for more complex distributions, several more advanced MCMC algorithms have been developed to improve on this. The Hamiltonian Monte Carlo (HMC) algorithm [18, 41] improve on the random walk nature of the Metropolis algorithm By generating proposals for the Metropolis update by simulating the dynamics of a Hamiltonian system. In practice, when using MCMC for inference in Bayesian neural networks, one would make use of either the HMC algorithm or one of its variants, such as the No-U-Turn Sampler (NUTS) [42], which is a more effective version of the HMC algorithm which also automatically tunes two key hyperparameters of the algorithm. However, the main problem of the MCMC algorithms remains, being that Monte Carlo methods turn challenging and infeasible in high dimensional applications [36]. The following section will look at variational inference, an alternative approach to approximate Bayesian inference.

2.4 Variational Inference (VI)

Let p be an intractable probability distribution. The main idea of variational inference methods [43] is to cast inference as an optimization problem over a class of tractable distributions Q in order to find a distribution $q_\phi \in Q$ that is as close as possible to p , so that we can use q_ϕ as an approximation of p. The approximation q_ϕ, called the *variational distribution*, is parameterized by a set of parameters ϕ that are optimized to minimize the distance between q_ϕ and p.

Evidence Lower Bound (ELBO)

Finding the best approximating variational distribution q^*_ϕ for a true posterior distribution $p(\theta|D)$ is formalized as minimizing the Kullback-Leibler-Divergance between the two distributions. *The Kullback Leibler Divergence (KL-Divergence)* [44] is a non-symmetric divergence that measures relative entropy or difference in information between two probability distributions. The KL-Divergence between the variational distribution q_ϕ and the posterior distribution $p(\theta|D)$ is defined as:

$$KL\left[q(\theta)||p(\theta|D)\right] = \int q(\theta) \log \frac{q(\theta)}{p(\theta|D)} d\theta = \mathbb{E}_{q(\theta)}\left[\log \frac{q(\theta)}{p(\theta|D)}\right] \tag{2.15}$$

By using the KL-Divergence as a measure of similarity, the Bayesian inference problem in equation 2.9 can be reformulated as an optimization problem where we try to find the best approximation $q_\phi^*(\theta) \in Q$ to the true posterior distribution $p(\theta|D)$ by minimizing the KL divergence between the two distributions.

$$q_\phi^*(\theta) = \arg\min_{q_\phi \in Q} \; KL\left[q(\theta)||p(\theta|D)\right] \tag{2.16}$$

One crucial problem with this objective is that the KL-Divergence requires the true posterior distribution $p(\theta|D)$ to be known, which is the distribution we are trying to approximate in the first place. To overcome this problem, we must instead derive an alternative objective. As shown in Blei et al. [45] the KL-Divergence can be manipulated to give the following objective:

$$\begin{aligned} KL\left[q(\theta)||p(\theta|D)\right] &= -\mathbb{E}_{q(\theta)}\left[\log \frac{p(\theta|D)}{q(\theta)}\right] \\ &= -\mathbb{E}_{q(\theta)}\left[\log \frac{p(\theta, D)}{p(D)q(\theta)}\right] \\ &= -\mathbb{E}_{q(\theta)}\left[\log \frac{p(\theta, D)}{q(\theta)} - \log p(D)\right] \\ &= -\mathbb{E}_{q(\theta)}\left[\log \frac{p(\theta, D)}{q(\theta)}\right] + \log p(D) \end{aligned} \tag{2.17}$$

The important thing to note here is that the $\log p(D)$ term is independent of the variational distribution q_ϕ and can therefore be ignored when minimizing the KL-Divergence. Another essential thing is that the first term in equation 2.17 presents a lower bound on the log-likelihood of the data D under the variational distribution q_ϕ. Since the KL-Divergence is always non-negative, minimizing the KL-Divergence is equivalent to maximizing the lower bound. The new cost function is called the Evidence Lower Bound (ELBO), also known as the variational free energy or variational lower bound [46].

$$\mathcal{L}_{ELBO} = \mathbb{E}_{q(\theta)}\left[\log \frac{p(\theta, D)}{q(\theta)}\right] \tag{2.18}$$

The new optimization problem is to maximize the ELBO with respect to the variational distribution q_ϕ.

$$q_\phi^*(\theta) = \arg\max_{q_\phi \in Q} \mathcal{L}_{ELBO} \tag{2.19}$$

More insight into the ELBO can be gained by rewriting the ELBO as shown in Blundell et al. [17]:

$$\mathcal{L}_{ELBO} = \mathbb{E}_{q(\theta)}\left[\log p(D|\theta)\right] - KL\left[q_\phi(\theta)||p(\theta)\right] \tag{2.20}$$

We can now see that the ELBO consists of a data-dependent term $\mathbb{E}_{q(\theta)}\left[\log p(D|\theta)\right]$ which will describe how well parameters sampled from the variational distribution q_ϕ fit the data D, as well as a regularizing prior-dependent term $KL(q_\phi||p(\theta))$ which will describe how close the variational distribution q_ϕ is to the prior distribution $p(\theta)$. The two terms are often referred to as the *likelihood cost* (data-dependent) and the *complexity cost* (prior-dependent) respectively [17]. We can view the ELBO as a trade-off between fitting the data while keeping the variational distribution close to the prior distribution.

While the ELBO is independent of the intractable posterior distribution $p(\theta|D)$, exactly computing it is still computationally intractable. Therefore, gradient descent and other optimization methods are used to approximate the true value. Standard gradient-based optimization techniques, such as Stochastic Gradient Descent (SGD) [30] or Adaptive Moment Estimation (Adam) [31] can then be used to maximize the ELBO for a set of parameters ϕ. The stochastic variational inference algorithm [20], is the SGD method applied to VI and is currently the most commonly used method for performing VI on Bayesian Neural Networks [24]. The SVI algorithm uses mini-batches of data to compute stochastic estimates of the ELBO and its gradients, which allows it to scale to larger models and datasets. Most implementations use few samples when evaluating the ELBO, which means that the gradient will be noisy at each iteration and slow to converge. Though SVI gives an attractive and scalable alternative to other sampling-based methods, it must still be adapted to deep learning.

2.4.1 Mean Field Variational Inference (MFVI)

When choosing a variational distribution q_ϕ, we want to find a distribution that is flexible enough to approximate the true posterior distribution $p(\theta|D)$ but also simple enough to be tractable. A common choice is to use a fully factorized Gaussian distribution, also known

as a mean-field distribution [47]. A Mean-field distribution follows the assumption that all latent variables are independent of each other, which greatly simplifies calculations. A mean-field Gaussian variational distribution would then be defined as:

$$q_\phi(\theta) = \prod_{i=1}^{N} q_\phi(\theta_i) = \prod_{i=1}^{N} \mathcal{N}(\mu_i, \sigma_i^2) \tag{2.21}$$

where μ_i and σ_i are the variational parameters for the i'th weight θ_i.

Because of the simplification in calculations, mean field variational inference is scalable to large models and datasets and has been successfully applied to neural networks [48, 20, 17]. However, mean field variational inference is also criticized for being too simple of an approximation, and especially for ignoring the correlations between different variables [47].

2.4.2 Bayes by Backprop (BBB)

Bayes by Backprop (BBB) [17] is a practical implementation of SVI, which leverages the *Reparameterization trick* [46, 49] to make the backpropagation algorithm work properly for stochastic weights. The main idea is to use a random variable $\epsilon \sim q(\epsilon)$ as a source of noise combined with a deterministic function $t(\phi, \epsilon)$ such that $\theta = t(\phi, \epsilon)$. The noise ϵ is sampled at each iteration, but it can be seen as a constant with regards to the parameters ϕ. Since the stochasticity has been removed from all other transformations; the gradient backpropagation algorithm works as usual for the variational parameters ϕ. The resulting training loop is, therefore, analogous to a normal neural network training loops, with the addition of sampling the noise ϵ at each iteration. By writing $\theta = t(\phi, \epsilon)$ and assuming $q(\epsilon)d\epsilon = q(\theta|\phi)d\theta$, *Blundell et al.*[17] prove that:

$$\begin{aligned} \frac{\partial}{\partial \phi} \mathbb{E}_{q(\theta|\phi)} \left[f(\theta, \phi) \right] &= \mathbb{E}_{q(\epsilon)} \left[\frac{\partial}{\partial \phi} f(t(\phi, \epsilon), \phi) \right] \\ &= \mathbb{E}_{q(\epsilon)} \left[\frac{\partial f(\theta, \phi)}{\partial \theta} \frac{\partial \theta}{\partial \phi} + \frac{\partial f(\theta, \phi)}{\partial \phi} \right] \end{aligned} \tag{2.22}$$

The BBB algorithm is derived by applying equation 2.22 to the ELBO optimization problem and provides an alternate approach to gradient estimation with respect to the variational parameters ϕ. The cost function itself is approximated with Monte Carlo

sampling by drawing i samples $\theta^i \sim q(\theta|\phi)$ and computing the average cost over all samples:

$$\mathcal{F}(\phi) \approx \sum_{i=1}^{N} \log q(\theta^i|\phi) - \log p(\theta^i) - \log p(D|\theta^i) \tag{2.23}$$

Algorithm 3 shows a general implementation of the BBB algorithm.

Algorithm 3 Bayes by Backprop [17]

Initialize variational parameters ϕ
repeat
  Sample $\epsilon \sim q(\epsilon)$
  $\theta = t(\phi, \epsilon)$
  $f(\theta, \phi) = \log q(\theta|\phi) - \log p(\theta)p(D|\theta)$
  $\nabla_\phi f = backprop_\phi(f)$
  $\phi = \phi - \alpha \nabla_\phi f$
until convergance

The objective function f corresponds to a stochastic estimate of the ELBO from a single weight sample θ^i, and will therefore be noisy.

In the case of a Gaussian variational posterior, the weights can be sampled from a unit Gaussian distribution and then shifted and scaled by parameters μ and σ respectively. In this case, the transform function t is given by $\theta = t(\phi, \epsilon) = \mu + \sigma\epsilon$. Since the σ parameter must always be positive, *Blundell et al.*[17] instead use the parameters ρ and μ, where $\sigma = \log(1 + exp(\rho))$. This results in the slightly modified transform function $\theta = t(\phi, \epsilon) = \mu + \log(1 + exp(\rho)) \circ \epsilon$, where $\circ$ denotes element-wise multiplication. Algorithm 4 shows the BBB algorithm adapted for a Gaussian variational posterior.

Algorithm 4 Bayes by Backprop for Gaussian variational posterior [17]

Initialize variational parameters ϕ
repeat
  Sample $\epsilon \sim \mathcal{N}(0, I)$
  $\theta = \mu + \log(1 + exp(\rho)) \circ \epsilon$
  $\phi = (\mu, \rho)$
  $f(\theta, \phi) = \log q(\theta|\phi) - \log p(\theta)p(D|\theta)$
  $\nabla_\mu f = \frac{\partial f(\theta,\phi)}{\partial \theta} + \frac{\partial f(\theta,\phi)}{\partial \mu}$
  $\nabla_\rho f = \frac{\partial f(\theta,\phi)}{\partial \theta} \frac{\epsilon}{1+exp(-\rho)} + \frac{\partial f(\theta,\phi)}{\partial \rho}$
  $\mu = \mu - \alpha \nabla_\mu f$
  $\rho = \rho - \alpha \nabla_\rho f$
until convergance

2.4.3 KL reweighting for minibatches

When performing gradient descent on larger datasets, it is common to separate the data into mini-batches as a compromise between fully stochastic gradient descent and full batch gradient descent. The training data D is split randomly into M mini-batches $D_1, D_2, ..., D_M$ for each epoch, and a gradient step is performed for each minibatch. However, the weights, which apply to the *compexity cost*, are only sampled once per epoch, while the total error cost is transmitted at each minibatch. To rectify this, *Graves (2011)* [48] proposes to reweight the complexity cost for minibatches. For minibatch $i = 1, 2, \ldots, M$:

$$\begin{aligned} &\pi_i = \frac{1}{M}, \\ &\mathcal{F}_i(D_i, \phi) = \pi_i KL(q_\phi || p(\theta)) - \mathbb{E}_{q_\phi}\left[\log p(D_i|\theta)\right]. \end{aligned} \tag{2.24}$$

Blundell et al. [17] proposed an alternative scaling factor for the complexity cost:

$$\pi_i = \frac{2^{M-i}}{2^M - 1} \tag{2.25}$$

By applying this scaling factor, the complexity cost is weighted such that the first few mini-batches are heavily reliant on the complexity cost, while the later minibatches almost exclusively rely on the likelihood cost. This is based on the assumption that while data is scarce, the complexity cost is more important, but as more data becomes available in the later mini-batches, the data should be more influential than the prior.

2.4.4 Local Reparameterization Trick

The local reparameterization trick [50] is an alternative gradient estimation technique for variational inference. Not to be confused with the reparameterization trick described in section 2.4.2, the local reparameterization trick builds on the idea that a factorized Gaussian posterior over the weights in a layer means that the posterior over the resulting activations is also a factorized Gaussian.

$$
\begin{aligned}
&q_\phi(\theta_{i,j}) = \mathcal{N}(\mu_{i,j}, \sigma^2_{i,j}) \Rightarrow q_\phi(a_{m,j}|x) = \mathcal{N}(\gamma_{m,j}, \delta_{m,j}), \\
&\text{where} \quad \gamma_{m,j} = \sum_{i=1}^{N} x_{m,i}\mu_{i,j} \quad \text{and} \quad \delta_{m,j} = \sum_{i=1}^{N} x^2_{m,i}\sigma^2_{i,j}.
\end{aligned} \tag{2.26}
$$

Using this assumption, instead of sampling each individual weight and then computing the activations, the pre-activations can be sampled directly from their implied Gaussian posterior distribution:

$$
a_{m,j} = \gamma_{m,j} + \sqrt{\delta_{m,j}}\epsilon_{m,j}, \quad \text{where} \quad \epsilon_{m,j} \sim \mathcal{N}(0,1). \tag{2.27}
$$

Since the activations are of a much lower dimensionality than the weights, this leads to significant computational gains. *Kingma et al. (2015)* [50] also highlights the fact that the local reparameterization trick reduces the variance of the gradient estimates, which leads to faster convergence.

Chapter 3

Motivation

This thesis will focus on exploring the use of Bayesian Neural Networks (BNNs) for regression and how the Gaussian likelihood observation model affects training and resulting posterior distribution of the model. We have a few key points of interest that we wish to explore in this thesis.

Firstly, we want to explore the Gaussian variance parameter σ. As a thought experiment, if we scale the variance parameter towards zero, we approach a point estimate similar to that of a standard neural network. Following this logic, we are interested in whether or not a low value for the variance parameter could cause a BNN to overfit the training data. Conversely, we are also interested in whether or not a high value for the variance parameter could cause a BNN to underfit the training data.

Secondly, we want to explore the effectiveness of inferring the variance parameter σ from the data. As the variance parameter is usually treated as a hyperparameter, we want to see if inferring it from the data can yield results on par with a well-specified fixed parameterization of the likelihood function.

Finally, we want to explore the effects of inferring the variance parameter σ for a misspecified model. In our experience with BNNs, we have found that misspecified models, especially smaller models with few parameters, struggle to express reasonable uncertainty estimates when trained using variational inference. We want to find confirmation of this behavior, as well as explore if inferring the variance parameter can help alleviate this issue.

3.1 Can extremes in the variance parameter cause over- and underfitting in BNNs?

When training a BNN regression model with a fixed Gaussian likelihood, the variance parameter must be set to a reasonable value. BNNs are, in general, very resilient to overfitting because of the implicit regularization introduced by setting a prior distribution over the weights. However, a poorly specified variance parameter can lead to behavior resembling both over- and underfitting.

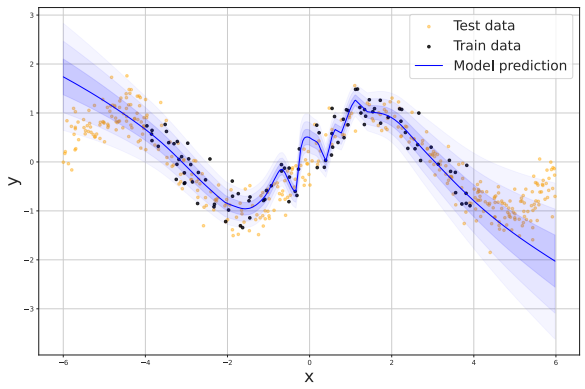
The posterior distribution of a BNN results from combining the prior distribution and the likelihood function. When using a Gaussian likelihood, the likelihood width (variance) directly impacts the shape of the posterior distribution.

If the Gaussian likelihood is too narrow, the model has very specific expectations about the data. Consequently, the posterior distribution will also be narrow, expressing a high confidence in the model parameters. This subsequently favors a small subset of the parameter space. The overconfidence in the learned parameters could cause the model to overemphasize the observed data and reduce the influence of the prior distribution. In such cases, the model becomes more prone to overfitting, especially if the data is noisy. The overconfident model could, for example, misinterpret noise as actual meaningful features of the data, leading to poor generalization.

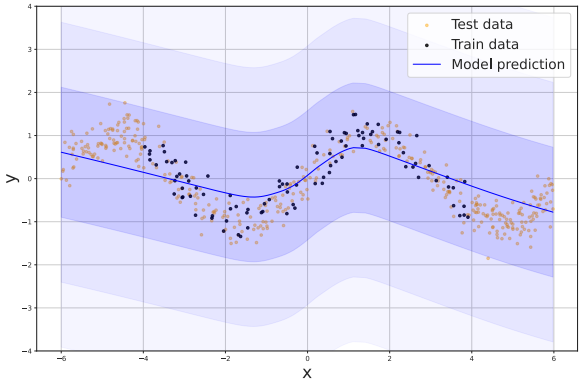
If the Gaussian likelihood is too wide, it results in a wider posterior distribution, which indicates high uncertainty in the model parameters. In this case, the observed data has less influence on the posterior, while the prior distribution has much more influence. The heightened uncertainty can prevent the model from converging to optimal parameter values, making it more prone to underfitting. Unlike the overconfident model, this underconfident model might misinterpret actual features of the data as noise, failing to learn the underlying patterns of the data. Even if it manages to learn the optimal parameters, its predictive distribution lacks the concentration necessary to make accurate predictions, which diminishes its overall performance. This behavior, as well as the behavior of the overconfident model, is illustrated in figure 3.1.

Determining an appropriate value for the variance parameter can often be a non-trivial task, often falling to the user's discretion. This becomes especially difficult when dealing with noisy or scarce data. In such cases, deciding what constitutes a "reasonable" value for the variance parameter is not always obvious. Making the wrong decision in such

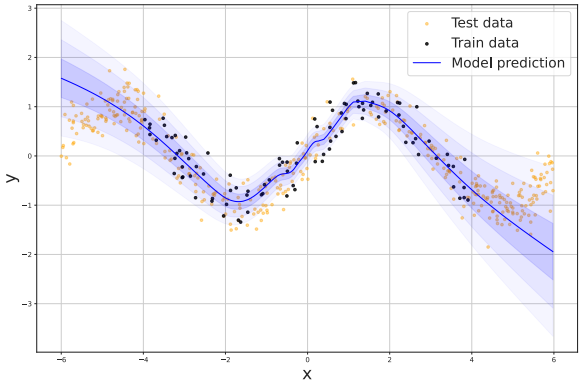
cases could lead to suboptimal model performance and unreliable uncertainty estimates. To address this issue, we wish to explore the effectiveness and reliability of a learnable variance parameter in the likelihood function. Our objective is to alleviate the burden of manual specification and instead let the model adapt organically to the inherent variance of the data. We expect this approach to produce equivalent or better results than a fixed variance parameter without the susceptibility to human error.



(a) Bnn overfitting due to poorly specified likelihood



(b) Bnn underfitting due to poorly specified likelihood



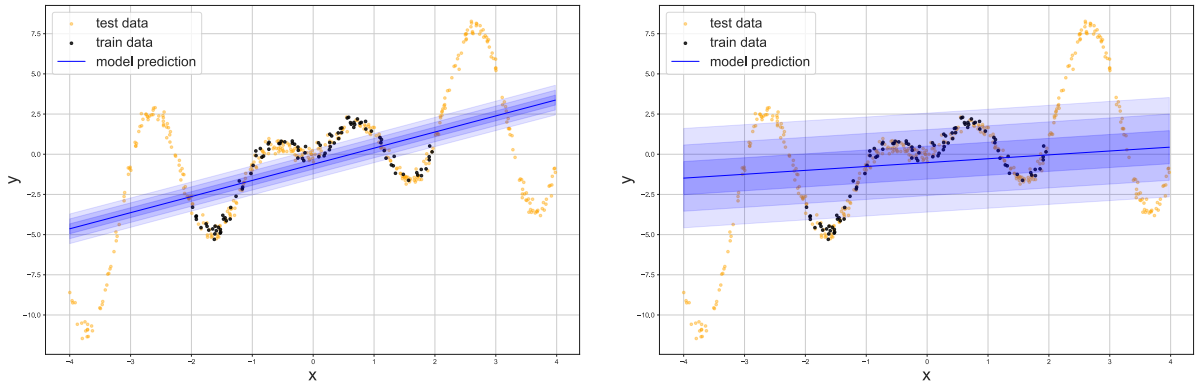
(c) Bnn with well specified likelihood

Figure 3.1: Illustration of over and underfitting-like behavior caused by a poorly specified likelihood. The three plots show the same Bayesian neural network trained using variational inference on the same dataset, but with different fixed variance parameters. The model in figure 3.1a has a fixed variance parameter that is too small and is showing signs of overfitting the data. The model in figure 3.1b has a fixed variance parameter that is too large and is showing signs of underfitting the data. The model in figure 3.1c has a fixed variance parameter that is well specified and shows no signs of over or underfitting.

3.2 Can inferring the variance parameter compensate for misspecified BNNs?

Misspecification of a machine learning model refers to the situation where the model is specified in a way that it is unable to express the true data-generating process. While "Knowing when we do not know" is the primary motivation behind using BNNs, the uncertainty in the posterior distribution becomes unreliable and often meaningless, given a grossly misspecified model. Misspecification leads to poor estimations of the posterior distribution, but this is not always obvious to detect. The desirable behavior of a misspecified model is to express high uncertainty in the posterior distribution, but this is not always the case. Sometimes, especially when using variational inference, the model will express overconfidence in its predictions, even for out-of-domain data. Introducing a learnable variance parameter in the likelihood function can help remedy this undesirable behavior. We expect that when a model is misspecified, our assumption of homoskedasticity will be broken. As a result, the learnable variance parameter will be able to scale to a high value to express this uncertainty.

As shown in figure 3.2, we found that when a linear Bayesian regression model is used to model non-linear data, the learnable variance parameter in the likelihood function learns to scale to a high value to fit the data. This results in a significantly increased uncertainty in the predictive distribution, which correctly expresses that the model is unable to learn the true data-generating process. We wish to explore this behavior further and see if it can be generalized to more complex models and data.



(a) Fixed variance

(b) Inferred variance

Figure 3.2: Linear Bayesian regression model trained on a non-linear dataset using variational inference. The model is misspecified, and with a fixed variance, the model presents overconfidence in its predictions. However, when the variance is inferred, it scales to a large value to express the uncertainty in the posterior distribution.

Chapter 4

Experimentation methodology

4.1 Datasets

We present a set of generated synthetic datasets that are used for various experiments. The datasets are a combination of a data-generating process and a homoscedastic Gaussian noise term. Combining different data-generating processes with different degrees of Gaussian noise, we can create datasets with varying degrees of complexity and innate uncertainty. Given a datagenerating process $f(x)$, we can generate a dataset D of size N as follows:

$$\epsilon \sim \mathcal{N}(0, std^2)$$
$$D = \{(x_i, y_i)\}_{i=1}^{N} = \{(x_i, f(x_i) + \epsilon)\}_{i=1}^{N}$$

All values of x are sampled from a uniform distribution with a given interval $[a, b]$:

$$x \sim \mathcal{U}(a, b)$$

To create validation and in-domain test sets, we sample N values from the same uniform distribution as the training set, and generate a dataset of size N using the same data-generating process and noise term as the training set. For out-of-domain test sets, we sample N values from a uniform distribution with a different interval $[2a, 2b]/[a, b]$

4.1.1 Sinusoidal datagenerating functions

We limit ourselves to using a set of sinusoidal data-generating functions, as they are simple to define and can be used to create datasets of varying complexity. Due to time constraints, it is best to thoroughly explore the results of a single type of function rather than trying to cover too many different types. We define the following sinusoidal data-generating functions:

Ten-dimensional sinusoidal (sin10)

This data-generating function takes a ten-dimensional input vector x and returns a one-dimensional output y:

$$x = [x_1, x_2, \ldots, x_{10}]$$

,

$$f(x) = 0x_1 + 6\sin(x_2 \cdot x_3) + 6\sin(x_4) + 6\sin(x_5 \cdot x_6) + 6\sin(x_7) + 6\sin(x_8 \cdot x_9) + 0x_{10}$$

We designed this function to be difficult to learn for small to medium-sized neural networks. The scaled sine functions will create large waves in the y space, which will be challenging to learn for a neural network with a small number of hidden units. The terms x_1 and x_{10} are nulled out and acts as input noise.

Multidiemensional sinusoidal (multisin)

This function is slightly less challenging than the *sin10* function, but the number of input dimensions is variable. This function will be helpful to test whether or not our results generalize to other similar datasets and higher dimensional spaces.

$$f(x) = \sum_{i=1}^{n-1} mask(i) \cdot 5 \cdot \sin(x_i + x_{i+1})$$

$$mask(i) = \begin{cases} 0 & \text{if } i \bmod 10 = 0 \text{ or } i+1 \bmod 10 = 0 \\ 1 & otherwise \end{cases}$$

The *mask* function makes every tenth input x_i exempt from the data-generating function and acts as input noise.

4.1.2 Generated Datasets

Using the data-generating functions and the pattern we described earlier, we generated a set of synthetic datasets with varying degrees of complexity. The datasets are described in table 4.1.

Dataset	Sample-size	Data-func	Noise std	X-dim	Y-dim	X-space
sin10-s03	10k	tendim-sinusoidal	0.3	10	1	[-2, 2]
multisin10-s05	10k	multisin-sinusoidal	0.5	10	1	[-3, 3]
multisin20-s05	20k	multisin-sinusoidal	0.5	20	1	[-3, 3]

Table 4.1: Table of generated datasets. Names of the datasets are reported together with the relevant data generation function, number of samples, and other relevant metadata.

4.2 Model specification

Functional Model

For all experiments, we implement Bayesian feed-forward neural networks of varying complexities. The models used for the various experiments vary in the number of layers and number of neurons per layer, and will be presented using the following notation:

$$\text{model} : \text{input} - h_1 - \cdots - h_n - \text{output}$$

As an example, a feed-forward Bayesian neural network with a 10-dimensional input, 3 hidden layers with 128 neurons each, and a 1-dimensional output will be presented as:

$$10\text{-}128\text{-}128\text{-}128\text{-}1$$

Activation Functions

For all models, we use the ReLU activation function for all hidden layers:

$$\text{ReLU}(x) = \max(0, x)$$

Variational Posterior

We define the variational posterior distribution over the weights of the neural network as a product of independent normal distributions with mean μ and standard deviation σ.

$$q_\phi(\theta) = \prod_{i=1}^{N} \mathcal{N}(\mu_i, \sigma_i^2)$$

Prior

We define the prior distribution over the weights of the neural network as a product of independent normal distributions with mean 0 and standard deviation 1.

$$p(\theta) = \prod_{i=1}^{N} \mathcal{N}(0, 1)$$

Considering the relative simplicity of our datasets and models, we believe the standard normal distribution to be a sufficient prior distribution. However, as mentioned in Fortuin et al. [21] the isotropic Gaussian prior is often misspecified for neural networks and might be suboptimal even for our simple models.

Likelihood

The model's likelihood function depends on the type of problem we are trying to solve. Considering a regression problem, we assume the data to be homoscedastic and the noise to be Gaussian distributed. Thus, we define the likelihood function as a simple Gaussian distribution:

$$y \sim \mathcal{N}(f(x, \theta), \sigma^2)$$

For some models the standard deviation σ is a fixed constant $\sigma \in \mathbb{N}$, while for others, it will be given a gamma distribution prior $\sigma \sim Gamma(\alpha, \beta)$. During training, the probability distribution of σ will be inferred from the data.

4.3 Model Inference

We train our models using the Stochastic Variational Inference algorithm combined with the Local Reparameterization Trick. Since our prior and posterior distributions are both Gaussians, we leverage the fact that the variational posterior distribution is a product of independent normal distributions to make use of the mean-field assumption and calculate the KL-divergence in closed form. We perform Mean Field Variational Inference to train a variational posterior distribution over the weights of the neural network. To reduce the variance of the gradient estimates, we make use of the local reparameterization trick mentioned in section 2.4.4.

Hyperparameters

For optimization, we use the Adam [31] optimizer. Following the advice of *Bingham et al.* [51] we use a small learning rate of 1e-4 and set the beta parameters to 0.95 and 0.999 respectively, to account for the increased stochasticity of a Bayesian neural network. They also mention that the variational posterior should be parameterized to have a low variance at initialization, as high variance in the elbo gradients at the beginning of optimization can lead to ending up in undesirable regions of the parameter space. We follow their advice and initialize the variational posterior to have a mean of 0 and a low standard deviation of 0.01.

We use a batch size of 512 for all experiments and train for 10000 epochs. When approximating the Gradient of the ELBO, we use 10 samples from the variational posterior distribution. This requires much more computing power as it requires a forward pass through the neural network for each sample. However, it greatly reduces the variance of the gradient estimates, which we found to sometimes be essential for the convergence of larger models.

4.4 Model Selection

Our approach to model selection might be unorthodox compared to the standard approach. Using mostly intuition and some trial and error, we found values for the hyperparameters that generally work well for all basic Bayesian neural networks. Using these

values and a fixed likelihood variance equal to the noise variance of the dataset, we train several models with different numbers of hidden layers and units and select the model with the lowest validation loss. This approach aims to find a decently performing model for each dataset when given an "optimal" parameterization of the likelihood. We can then use these models as a baseline to compare against when performing our experiments.

Using the abovementioned approach, we use model selection to find a well-performing baseline model for each dataset. The final models for each dataset are reported in table 4.2.

Dataset	Model name	Architecture	Likelihood σ
sin10-s03	sin10-3x256-s03	10-256-256-256-1	0.3
multisin10-s05	multisin10-3x64-s05	10-64-64-64-1	0.5
multisin20-s05	multisin20-3x512-s05	20-512-512-512-1	0.5

Table 4.2: Table of selected baseline models for each dataset.

4.5 Evaluation Metrics

We employ a set of metrics to evaluate our models' performance on the various datasets. For all models and experiments, we record the Root Mean Squared Error (RMSE), Mean Absolute Error (MAE) and Log Likelihood (LL) of the model predictions on the train, validation, in-domain test, and out-of-domain test sets.

In addition to the standard evaluation metrics mentioned above, we also need some metrics to evaluate the uncertainty of our models. We use the following metrics to evaluate the uncertainty of our models:

Mean, Min, and Max Weight Uncertainty

Evaluating and quantifying the uncertainty of a Bayesian neural network is a challenging task, which grows exponentially more difficult with the increasing complexity of the model. While getting a complete picture of the uncertainty of our Bayesian neural network is impossible without a full posterior distribution over the weights, we can still get some idea of overall uncertainty by summarizing the mean, min, and max values of the standard deviations of the variational posterior distribution over the weights. We can then use these

values to compare the uncertainty of different models. Since our variational posterior is a product of independent normal distributions, we can look at the standard deviation of each weight independently and summarize the uncertainty of the model by looking at the mean, min, and max values of these standard deviations. These metrics are presented as *Mean Weight SD*, *Min Weight SD*, and *Max Weight SD*.

Mean, Min, and Max Prediction Uncertainty

In addition to evaluating the uncertainty of the model weights, we can gain further insight by evaluating the uncertainty of the model's predictions. In contrast with the weight uncertainty, we can get a much better picture of the predictive uncertainty for each data point, as our output predictive distribution is 1-dimensional Gaussian distribution. However, due to the high dimensional input space, plotting the predictive distribution for each data point is impossible. Instead, similarly to the weight uncertainty, we summarize the uncertainty of the model by looking at the mean, min, and max values for the standard deviation of the predictive distribution to get a general idea of the overall uncertainty of the model on a given dataset. These metrics are presented as *Mean Predictive SD*, *Min Predictive SD*, and *Max Predictive SD*.

4.6 Implementation Details

Hardware

All tests were run on an NVIDIA A100 SXM4 GPU with 80GB VRAM.

Software and Libraries

All the code used for our experiments is available in our public GitHub repository. [1] The code is written in Python 3.9, and an environment file is provided to reproduce the anaconda environment used for the experiments. We employed a set of different software libraries to implement and run our Bayesian neural networks. The most important libraries are listed below.

[1] `https://github.com/alvarhonsi/master-pipeline`

Pytorch [52] is an open-source machine-learning framework used for many different machine learning tasks, like computer vision and natural language processing. Being an optimized tensor library for deep learning, Pytorch provides two important high-level features: Tensor computing with strong acceleration via GPUs as well as deep neural networks built on a tape-based autograd system.

Pyro [53] is a probabilistic programming language built on Python as a platform for developing advanced probabilistic models in AI research. Pyro provides flexible variational inference algorithms, MCMC algorithms, and an extensive library of probability distributions built on top of PyTorch. To accommodate more complex or model-specific algorithms, Pyro also includes the Poutine library, which includes composable handlers for modifying the behavior of probabilistic programs.

TyXe [54] is a Bayesian neural network library built on top of pytorch and Pyro. It provides a simple interface for building and training Bayesian neural networks. TyXe simplifies KL-reweighting for networks with multiple layers, and most importantly, it provides implementations of essential BNN-specific event handlers for Pyro, such as the *Local Reparameterization Trick*. It is, however, worth noting that TyXe is still in early development, has little to no documentation, and is not yet a stable library. As such, we have had to make some minor modifications to the library to accommodate our needs. Most importantly, we had to alter the homoskedastic Gaussian likelihood and BNN implementations to make the inferred variance parameter work as intended. The relevant changes are in the appendix A.

Chapter 5

Results

In this chapter, we present our experiments and their results. Using the datasets and baseline models described in chapter 4, we perform two studies in the hopes of gaining some insight into how subtle changes in the specification of our Gaussian likelihood function impacts the training process and the resulting posterior distribution.

5.1 Study 1: Can extremes in the variance parameter cause over- and underfitting in BNNs?

From our model selection in chapter 4, we have a set of well-performing baseline models for each dataset. Using the baseline models as a starting point, we train several iterations of the same model but with differently parameterized Gaussian likelihoods. Our aim with this study is to understand more about how deviations in the variance of the Gaussian likelihood function can impact the training process and the resulting posterior distribution. Bayesian Neural Networks are intricate and complex models with many moving parts. This makes observing the impact of a single change in the model specification difficult as combinations of different factors come into play. While observing the effects of the likelihood in a vacuum is not possible, we try to create a stable baseline by using the same architecture, hyperparameters, and priors for all models.

As mentioned in chapter 3, we expect that by making the Gaussian likelihood either too narrow or too wide, we will observe overfitting and underfitting behavior in our model. In order to test this hypothesis, we specify models with small and large fixed likelihood

variances. We make the likelihood variance lower or higher than the actual noise in the data by a factor of 10. If our hypothesis has merit, we should observe some clear overfitting and underfitting behavior in the models. We have a secondary objective of exploring the effectiveness of inferring the likelihood variance from the data. We expect that inferring the likelihood variance should yield results close to or on par with the "optimal" likelihood variance in the baseline model.

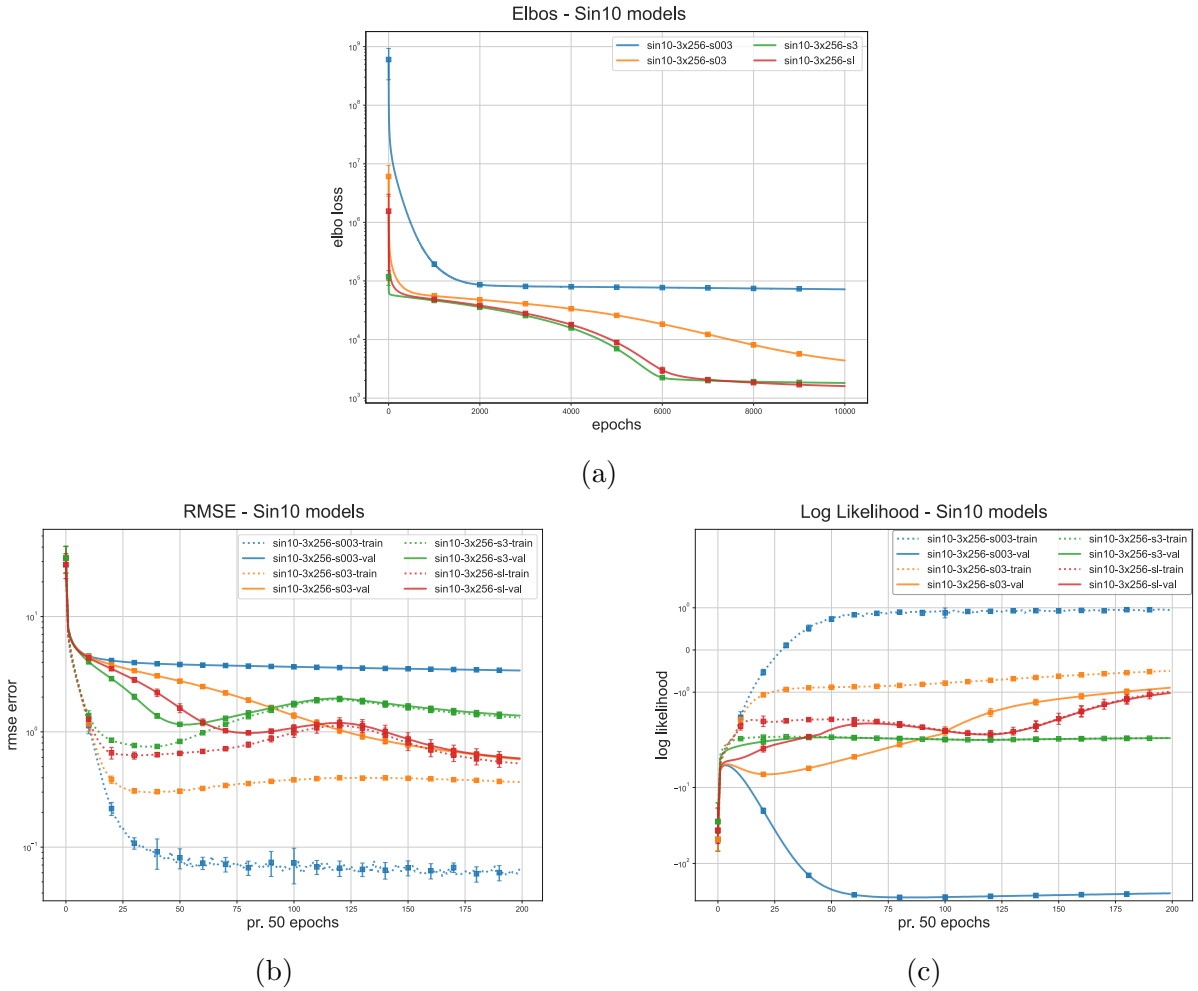
We define four models for each dataset: one with a low fixed variance, one with an "optimal" fixed variance, one with a high fixed variance, and lastly, one with an inferred variance. The models are defined in table 5.1. Our results for each dataset are presented in the following sections.

Dataset	Model name	Architecture	Likelihood σ
sin10-s03	sin10-s003	10-256-256-256-1	0.03
	sin10-s03		0.3
	sin10-s3		3.0
	sin10-sl		$\sim$ Gamma(1.0, 1.0)
multisin10-s05	multisin10-s005	10-64-64-64-1	0.05
	multisin10-s05		0.5
	multisin10-s5		5.0
	multisin10-sl		$\sim$ Gamma(1.0, 1.0)
multisin20-s05	multisin20-s005	20-512-512-512-1	0.05
	multisin20-s05		0.5
	multisin20-s5		5.0
	multisin20-sl		$\sim$ Gamma(1.0, 1.0)

Table 5.1: Table of models used in the first study. The table shows the models used for each dataset. The models are defined by their name and architecture, as well as the variance of the Gaussian likelihood function. The notation $\sim$ Gamma(1.0, 1.0) indicates that the model sets a prior distribution $Gamma(1.0, 1.0)$ over the likelihood parameter σ and infers its posterior distribution from the data.

5.1.1 Results for the *sin10-s03* dataset

This section presents our results for the *sin10-s03* dataset. We present the training curves for the four models in figure 5.1. We present further evaluation metrics in table 5.2. Our metrics for summarized predictive uncertainty are presented in table 5.3 and our metrics for summarized weight uncertainty are presented in table 5.4.



(a)

(b)

(c)

Figure 5.1: Study 1 - Training curves for models trained on the *sin10-s03* dataset. Figure 5.1a shows the elbo loss at each epoch. Figures 5.1b and 5.1c show the RMSE and log-likelihood metrics, respectively, for both train and validation data, recorded every 50 epochs. The curves are averaged over 10 independent runs, and the error bars show a confidence interval of $2 * SD$ (2 times the standard deviation) to show the differences between random initializations. The y-axis is log-scaled.

(a) Train data

Model	Likelihood σ	RMSE	MAE	LL
sin10-3x256-s003	0.03±0.0	0.06±0.02	0.05±0.02	0.95±0.07
sin10-3x256-s03	0.3±0.0	0.37±0.02	0.29±0.02	-0.5±0.04
sin10-3x256-s3	3.0±0.0	1.31±0.1	0.96±0.07	-2.23±0.02
sin10-3x256-sl	0.75±0.15	0.53±0.11	0.42±0.08	-0.98±0.19

(b) In domain data

Model	Likelihood σ	RMSE	MAE	LL
sin10-3x256-s003	0.03±0.0	3.4±0.16	2.59±0.13	-241.82±26.15
sin10-3x256-s03	0.3±0.0	0.58±0.06	0.45±0.04	-0.88±0.11
sin10-3x256-s3	3.0±0.0	1.4±0.1	1.0±0.07	-2.24±0.02
sin10-3x256-sl	0.75±0.15	0.59±0.12	0.47±0.09	-1.02±0.2

(c) Out of domain data

Model	Likelihood σ	RMSE	MAE	LL
sin10-3x256-s003	0.03±0.0	18.56±0.62	14.87±0.48	-809.1±147.68
sin10-3x256-s03	0.3±0.0	34.86±3.53	28.65±3.03	-28.01±5.05
sin10-3x256-s3	3.0±0.0	18.4±1.76	14.77±1.39	-4.72±0.19
sin10-3x256-sl	0.75±0.15	31.45±5.8	26.33±4.51	-6.63±1.63

Table 5.2: Study 1 - Table of results for models trained on the *sin10-s03* dataset. The table shows the Root Mean Squared Error (RMSE), Mean Absolute Error (MAE) and log-likelihood (LL) evaluation metrics for each model. Subtables 5.2a, 5.2b and 5.2c show the results for the train, in-domain test, and out-of-domain test data, respectively. The results are averaged over 10 independent runs and presented as the mean value $\pm\ 2 * SD$ (2 times the standard deviation) in order to highlight the differences between random initializations.

(a) Train data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
sin10-3x256-s003	0.03±0.0	0.14±0.0	0.07±0.01	0.26±0.04
sin10-3x256-s03	0.3±0.0	0.47±0.03	0.32±0.0	1.3±0.14
sin10-3x256-s3	3.0±0.0	3.51±0.05	3.03±0.02	5.94±0.57
sin10-3x256-sl	0.75±0.15	0.9±0.17	0.76±0.14	2.04±0.35

(b) In domain data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
sin10-3x256-s003	0.03±0.0	0.15±0.0	0.07±0.01	0.39±0.15
sin10-3x256-s03	0.3±0.0	0.47±0.03	0.32±0.01	1.28±0.19
sin10-3x256-s3	3.0±0.0	3.51±0.05	3.04±0.03	5.9±0.26
sin10-3x256-sl	0.75±0.15	0.91±0.18	0.76±0.14	2.03±0.33

(c) Out of domain data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
sin10-3x256-s003	0.03±0.0	0.48±0.04	0.29±0.04	2.05±1.23
sin10-3x256-s03	0.3±0.0	5.01±0.46	2.57±0.27	8.51±1.03
sin10-3x256-s3	3.0±0.0	41.66±10.28	12.88±2.16	104.8±29.43
sin10-3x256-sl	0.75±0.15	14.11±2.39	5.22±1.19	35.44±13.8

Table 5.3: Study 1 - Table of summarized predictive uncertainty for models trained on the *sin10-s03* dataset. Tables 5.3a, 5.3b and 5.3c show the results for the train, in-domain test, and out-of-domain test data, respectively. The tables show the mean uncertainty of all predictive distributions (Mean Predictive SD) for the given dataset, as well as the minimum uncertainty (Min Predictive SD) and maximum uncertainty (Max Predictive SD) of all predictive distributions for the given dataset. The results are averaged over 10 independent runs and presented as the mean value $\pm\ 2 * SD$ (2 times the standard deviation) to highlight the differences between random initializations.

(a) Model weight data

Model	Mean Weight SD	Min Weight SD	Max Weight SD
sin10-3x256-s003	0.05±0.02	0.0±0.0	1.01±0.0
sin10-3x256-s03	0.87±0.02	0.0±0.0	1.02±0.02
sin10-3x256-s3	0.97±0.0	0.0±0.0	1.0±0.0
sin10-3x256-sl	0.96±0.0	0.0±0.0	1.01±0.01

Table 5.4: Study 1 - Table of summarized weight uncertainty for models trained on the *sin10-s03* dataset. The table shows the mean uncertainty of all weight distributions in the models (Mean Weight SD), as well as the minimum uncertainty (Min Weight SD) and maximum uncertainty (Max Weight SD) of all weight distributions in the models. The results are averaged over 10 independent runs and presented as the mean value $\pm$ $2{*}SD$ (2 times the standard deviation) to highlight the differences between random initializations.

5.1.2 Results for the *multisin10-s05* dataset

This section presents our results for the *multisin10-s05* dataset. We present the training curves for the four models in figure 5.2. We present further evaluation metrics in table 5.5. Our metrics for summarized predictive uncertainty are presented in table 5.6 and our metrics for summarized weight uncertainty are presented in table 5.7.

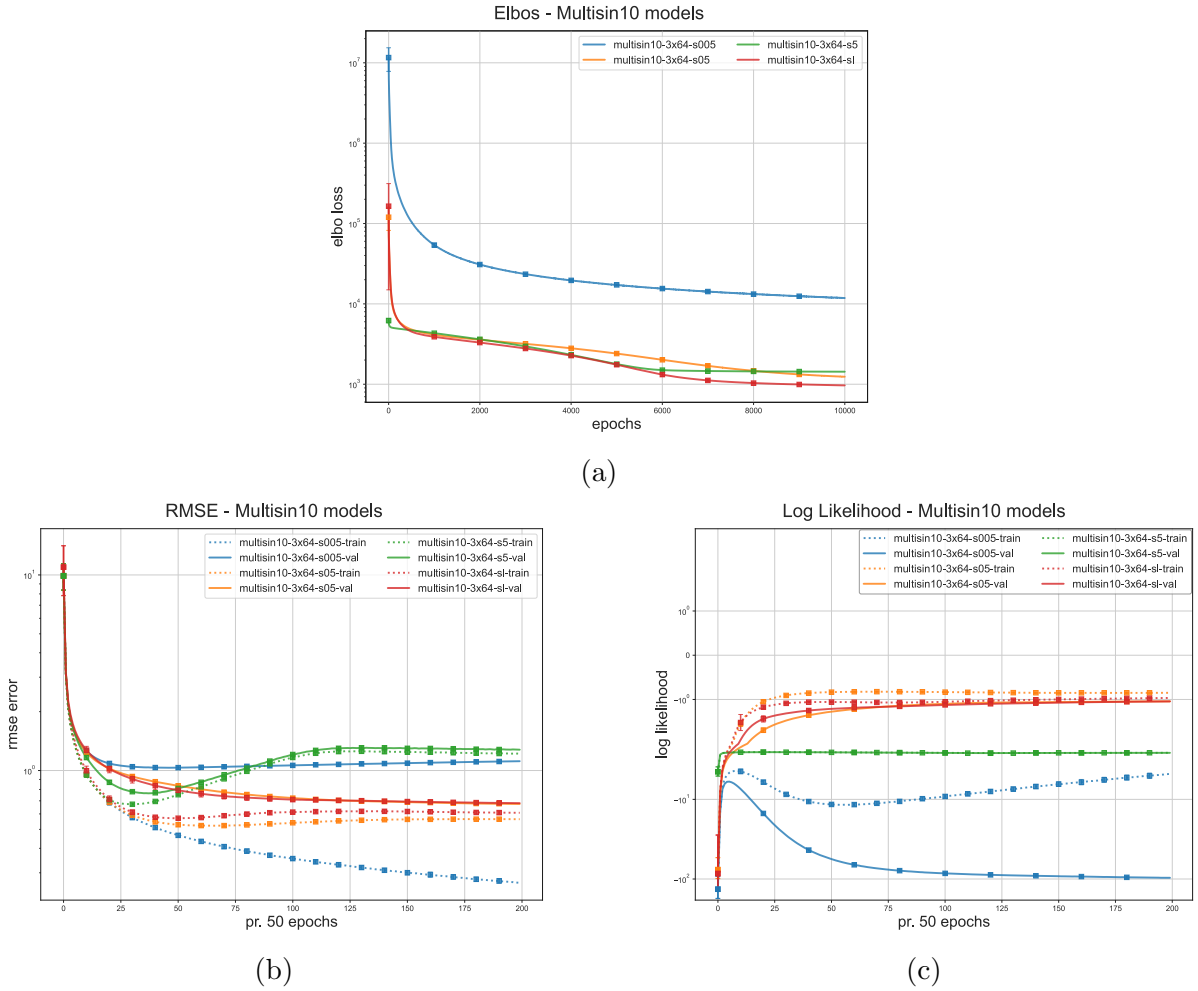


Figure 5.2: Study 1 - Training curves for models trained on the *multisin10-s05* dataset. Figure 5.2a shows the elbo loss at each epoch. Figures 5.2b and 5.2c show the RMSE and log-likelihood metrics, respectively, for both train and validation data, recorded every 50 epochs. The curves are averaged over 10 independent runs, and the error bars show a confidence interval of $2 * SD$ (2 times the standard deviation) to show the differences between random initializations. The y-axis is log-scaled.

(a) Train data

Model	Likelihood σ	RMSE	MAE	LL
multisin10-3x64-s005	0.05±0.0	0.27±0.01	0.21±0.01	-4.72±0.74
multisin10-3x64-s05	0.5±0.0	0.56±0.01	0.45±0.01	-0.85±0.02
multisin10-3x64-s5	5.0±0.0	1.21±0.04	0.94±0.03	-2.6±0.01
multisin10-3x64-sl	0.69±0.05	0.61±0.02	0.48±0.02	-0.97±0.03

(b) In domain data

Model	Likelihood σ	RMSE	MAE	LL
multisin10-3x64-s005	0.05±0.0	1.11±0.06	0.87±0.04	-94.23±8.42
multisin10-3x64-s05	0.5±0.0	0.68±0.02	0.54±0.02	-1.05±0.04
multisin10-3x64-s5	5.0±0.0	1.28±0.04	0.99±0.03	-2.6±0.01
multisin10-3x64-sl	0.69±0.05	0.69±0.02	0.54±0.01	-1.05±0.02

(c) Out of domain data

Model	Likelihood σ	RMSE	MAE	LL
multisin10-3x64-s005	0.05±0.0	11.87±0.31	9.47±0.23	-1750.7±87.14
multisin10-3x64-s05	0.5±0.0	11.96±0.41	9.56±0.33	-33.61±5.05
multisin10-3x64-s5	5.0±0.0	9.48±0.13	7.61±0.11	-3.85±0.04
multisin10-3x64-sl	0.69±0.05	11.66±0.29	9.32±0.25	-17.3±6.98

Table 5.5: Study 1 - Table of results for models trained on the *multisin10-s05* dataset. The table shows the Root Mean Squared Error (RMSE), Mean Absolute Error (MAE) and log-likelihood (LL) evaluation metrics for each model. Subtables 5.5a, 5.5b and 5.5c show the results for the train, in-domain test, and out-of-domain test data, respectively. The results are averaged over 10 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) to highlight the differences between random initializations.

(a) Train data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
multisin10-3x64-s005	0.05±0.0	0.08±0.0	0.05±0.0	0.19±0.06
multisin10-3x64-s05	0.5±0.0	0.58±0.01	0.5±0.0	0.86±0.08
multisin10-3x64-s5	5.0±0.0	5.24±0.03	4.99±0.02	5.84±0.25
multisin10-3x64-sl	0.69±0.05	0.76±0.02	0.69±0.01	1.08±0.1

(b) In domain data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
multisin10-3x64-s005	0.05±0.0	0.08±0.0	0.05±0.0	0.28±0.34
multisin10-3x64-s05	0.5±0.0	0.59±0.01	0.5±0.0	0.9±0.08
multisin10-3x64-s5	5.0±0.0	5.24±0.03	5.01±0.03	5.8±0.32
multisin10-3x64-sl	0.69±0.05	0.76±0.02	0.69±0.01	1.14±0.1

(c) Out of domain data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
multisin10-3x64-s005	0.05±0.0	0.2±0.01	0.11±0.01	1.48±1.87
multisin10-3x64-s05	0.5±0.0	1.46±0.13	0.86±0.11	2.45±0.46
multisin10-3x64-s5	5.0±0.0	6.47±0.24	5.8±0.08	8.43±1.63
multisin10-3x64-sl	0.69±0.05	2.06±0.5	1.24±0.18	4.42±2.48

Table 5.6: Study 1 - Table of summarized predictive uncertainty for models trained on the *multisin10-s05* dataset. Tables 5.6a, 5.6b and 5.6c show the results for the train, in-domain test, and out-of-domain test data, respectively. The tables show the mean uncertainty of all predictive distributions (Mean Predictive SD) for the given dataset, as well as the minimum uncertainty (Min Predictive SD) and maximum uncertainty (Max Predictive SD) of all predictive distributions for the given dataset. The results are averaged over 10 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) to highlight the differences between random initializations.

(a) Model weight data

Model	Mean Weight SD	Min Weight SD	Max Weight SD
multisin10-3x64-s005	0.0±0.0	0.0±0.0	0.73±0.69
multisin10-3x64-s05	0.69±0.04	0.0±0.0	1.0±0.0
multisin10-3x64-s5	0.9±0.01	0.0±0.0	1.0±0.0
multisin10-3x64-sl	0.81±0.02	0.0±0.0	1.0±0.0

Table 5.7: Study 1 - Table of summarized weight uncertainty for models trained on the *multisin10-s05* dataset. The table shows the mean uncertainty of all weight distributions in the models (Mean Weight SD), as well as the minimum uncertainty (Min Weight SD) and maximum uncertainty (Max Weight SD) of all weight distributions in the models. The results are averaged over 10 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) in order to highlight the differences between random initializations.

5.1.3 Results for the *multisin20-s05* dataset

This section presents our results for the *multisin20-s05* dataset. We present the training curves for the four models in figure 5.3. We present further evaluation metrics in table 5.8. Our metrics for summarized predictive uncertainty are presented in table 5.9 and our metrics for summarized weight uncertainty are presented in table 5.10.

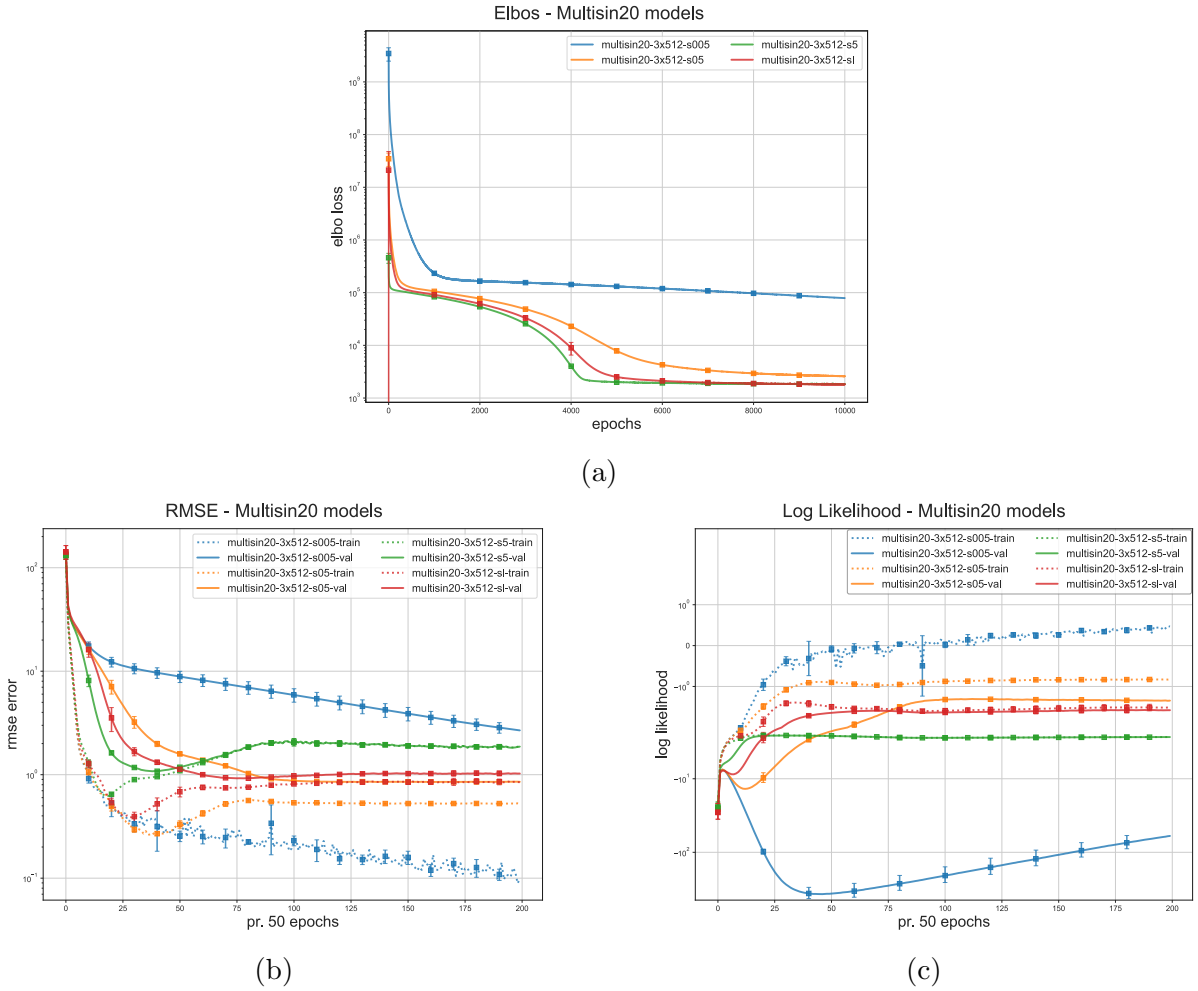


Figure 5.3: Study 1 - Training curves for models trained on the *multisin20-s05* dataset. Figure 5.3a shows the elbo loss at each epoch. Figures 5.3b and 5.3c show the RMSE and log-likelihood metrics, respectively, for both train and validation data, recorded every 50 epochs. The curves are averaged over 5 independent runs, and the error bars show a confidence interval of $2 * SD$ (2 times the standard deviation) to show the differences between random initializations. The y-axis is log-scaled.

(a) Train data

Model	Likelihood σ	RMSE	MAE	LL
multisin20-3x512-s005	0.05±0.0	0.1±0.04	0.08±0.03	0.46±0.1
multisin20-3x512-s05	0.5±0.0	0.52±0.01	0.41±0.01	-0.82±0.02
multisin20-3x512-s5	5.0±0.0	1.85±0.19	1.37±0.12	-2.73±0.02
multisin20-3x512-sl	1.29±0.23	0.84±0.1	0.67±0.07	-1.52±0.16

(b) In domain data

Model	Likelihood σ	RMSE	MAE	LL
multisin20-3x512-s005	0.05±0.0	2.68±0.56	2.1±0.46	-58.28±20.12
multisin20-3x512-s05	0.5±0.0	0.86±0.02	0.69±0.02	-1.34±0.03
multisin20-3x512-s5	5.0±0.0	1.98±0.2	1.47±0.13	-2.73±0.03
multisin20-3x512-sl	1.29±0.23	1.04±0.11	0.83±0.08	-1.6±0.14

(c) Out of domain data

Model	Likelihood σ	RMSE	MAE	LL
multisin20-3x512-s005	0.05±0.0	16.06±0.63	12.85±0.53	-283.74±9.08
multisin20-3x512-s05	0.5±0.0	15.23±2.23	12.17±1.79	-5.11±0.13
multisin20-3x512-s5	5.0±0.0	14.18±0.18	11.35±0.13	-4.39±0.12
multisin20-3x512-sl	1.29±0.23	15.48±2.1	12.33±1.57	-4.73±1.04

Table 5.8: Study 1 - Table of results for models trained on the *multisin20-s05* dataset. The table shows the Root Mean Squared Error (RMSE), Mean Absolute Error (MAE) and log-likelihood (LL) evaluation metrics for each model. Subtables 5.8a, 5.8b and 5.8c show the results for the train, in-domain test, and out-of-domain test data, respectively. The results are averaged over 5 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) in order to highlight the differences between random initializations.

(a) Train data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
multisin20-3x512-s005	0.05±0.0	0.23±0.01	0.11±0.01	0.57±0.15
multisin20-3x512-s05	0.5±0.0	0.67±0.01	0.53±0.01	1.59±0.19
multisin20-3x512-s5	5.0±0.0	5.8±0.11	5.15±0.09	14.31±4.69
multisin20-3x512-sl	1.29±0.23	1.6±0.32	1.33±0.25	5.23±2.64

(b) In domain data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
multisin20-3x512-s005	0.05±0.0	0.25±0.01	0.13±0.02	0.6±0.12
multisin20-3x512-s05	0.5±0.0	0.67±0.0	0.53±0.01	1.45±0.63
multisin20-3x512-s5	5.0±0.0	5.85±0.13	5.18±0.11	14.17±5.55
multisin20-3x512-sl	1.29±0.23	1.61±0.28	1.34±0.26	4.21±2.72

(c) Out of domain data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
multisin20-3x512-s005	0.05±0.0	0.7±0.02	0.41±0.03	1.79±0.83
multisin20-3x512-s05	0.5±0.0	66.9±8.6	19.67±4.64	175.7±42.53
multisin20-3x512-s5	5.0±0.0	9.24±0.36	6.55±0.64	36.72±19.99
multisin20-3x512-sl	1.29±0.23	41.86±87.11	14.37±30.63	142.81±273.18

Table 5.9: Study 1 - Table of summarized predictive uncertainty for models trained on the *multisin20-s03* dataset. Tables 5.9a, 5.9b and 5.9c show the results for the train, in-domain test, and out-of-domain test data, respectively. The tables show the mean uncertainty of all predictive distributions (Mean Predictive SD) for the given dataset, as well as the minimum uncertainty (Min Predictive SD) and maximum uncertainty (Max Predictive SD) of all predictive distributions for the given dataset. The results are averaged over 5 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) in order to highlight the differences between random initializations.

(a) Model weight data

Model	Mean Weight SD	Min Weight SD	Max Weight SD
multisin20-3x512-s005	0.32±0.07	0.0±0.0	1.01±0.0
multisin20-3x512-s05	0.97±0.0	0.0±0.0	1.01±0.0
multisin20-3x512-s5	0.99±0.0	0.0±0.0	1.0±0.0
multisin20-3x512-sl	0.98±0.0	0.0±0.0	1.01±0.0

Table 5.10: Study 1 - Table of summarized weight uncertainty for models trained on the *multisin20-s03* dataset. The table shows the mean uncertainty of all weight distributions in the models (Mean Weight SD), as well as the minimum uncertainty (Min Weight SD) and maximum uncertainty (Max Weight SD) of all weight distributions in the models. The results are averaged over 5 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) in order to highlight the differences between random initializations.

5.2 Study 2: Can inferring the variance parameter compensate for misspecified BNNs?

This study explores how inferring the likelihood variance can help compensate for misspecified models. As explained in chapter 3, we often find that misspecified Bayesian neural networks exhibit overconfident uncertainty estimates and an inability to recognize their misspecification. In the case of a severely misspecified model, we ideally want it to express high uncertainty so that we at least know not to trust its predictions. However, as we have experienced, and as shown in the linear Bayesian regression example in chapter 3, misspecified models often express low uncertainty and high confidence in their predictions. This is undesirable in a Bayesian model, where we want to produce high-quality uncertainty estimates. This is where we believe inferring the likelihood variance can help. We believe inferring the likelihood variance can help compensate for misspecified models by allowing the model to express higher uncertainty in its predictions. Our goals for this study are, therefore, twofold. Firstly, we want confirmation that misspecified models express overconfident uncertainty estimates. Secondly, we want to explore if inferring the likelihood variance can help compensate for misspecified models.

We use the same datasets and baseline models from chapter 4 and the previous study. Starting with the baseline models, we attempt to specify increasingly misspecified models by incrementally decreasing the number of parameters and depth of the Bayesian neural network, ending with a linear Bayesian regression model. For each model architecture,

we specify two models, one with a fixed "optimal" likelihood variance and one with an inferred likelihood variance. We believe we can gain some insight by comparing the performance of the two likelihoods on increasingly misspecified models. Our results for each dataset are presented in the following sections. The models are defined in table 5.11. Our results for each dataset are presented in the following sections.

Dataset	Model name	Architecture	Likelihood σ
sin10-s03	sin10-3x256-s03	10-256-256-256-1	0.3
	sin10-3x256-sl	10-256-256-256-1	$\sim$ Gamma(1.0, 1.0)
	sin10-2x128-s03	10-128-128-1	0.3
	sin10-2x128-sl	10-128-128-1	$\sim$ Gamma(1.0, 1.0)
	sin10-1x64-s03	10-64-1	0.3
	sin10-1x64-sl	10-64-1	$\sim$ Gamma(1.0, 1.0)
	sin10-linear-s03	10-1	0.3
	sin10-linear-sl	10-1	$\sim$ Gamma(1.0, 1.0)
multisin10-s05	multisin10-3x64-s03	10-64-64-64-1	0.5
	multisin10-3x64-sl	10-64-64-64-1	$\sim$ Gamma(1.0, 1.0)
	multisin10-2x32-s03	10-32-32-1	0.5
	multisin10-2x32-sl	10-32-32-1	$\sim$ Gamma(1.0, 1.0)
	multisin10-1x16-s03	10-16-1	0.5
	multisin10-1x16-sl	10-16-1	$\sim$ Gamma(1.0, 1.0)
	multisin10-linear-s03	10-1	0.5
	multisin10-linear-sl	10-1	$\sim$ Gamma(1.0, 1.0)
multisin20-s05	multisin20-3x512-s03	20-512-512-512-1	0.5
	multisin20-3x512-sl	20-512-512-512-1	$\sim$ Gamma(1.0, 1.0)
	multisin20-2x256-s03	20-256-256-1	0.5
	multisin20-2x256-sl	20-256-256-1	$\sim$ Gamma(1.0, 1.0)
	multisin20-1x128-s03	20-128-1	0.5
	multisin20-1x128-sl	20-128-1	$\sim$ Gamma(1.0, 1.0)
	multisin20-linear-s03	20-1	0.5
	multisin20-linear-sl	20-1	$\sim$ Gamma(1.0, 1.0)

Table 5.11: Table of models used in the second study. The table shows the models used for each dataset. The models are defined by their name and architecture, as well as the variance of the Gaussian likelihood function. The notation $\sim$ Gamma(1.0, 1.0) indicates that the model sets a prior distribution $Gamma(1.0, 1.0)$ over the likelihood parameter σ and infers its posterior distribution from the data.

5.2.1 Results for the *sin10-s03* dataset

This section presents our results for the *sin10-s03* dataset. We start by presenting the training curves for the models in figure 5.4. We present further evaluation metrics in

table 5.12. Our metrics for summarized predictive uncertainty are presented in table 5.13 and our metrics for summarized weight uncertainty are presented in table 5.14.

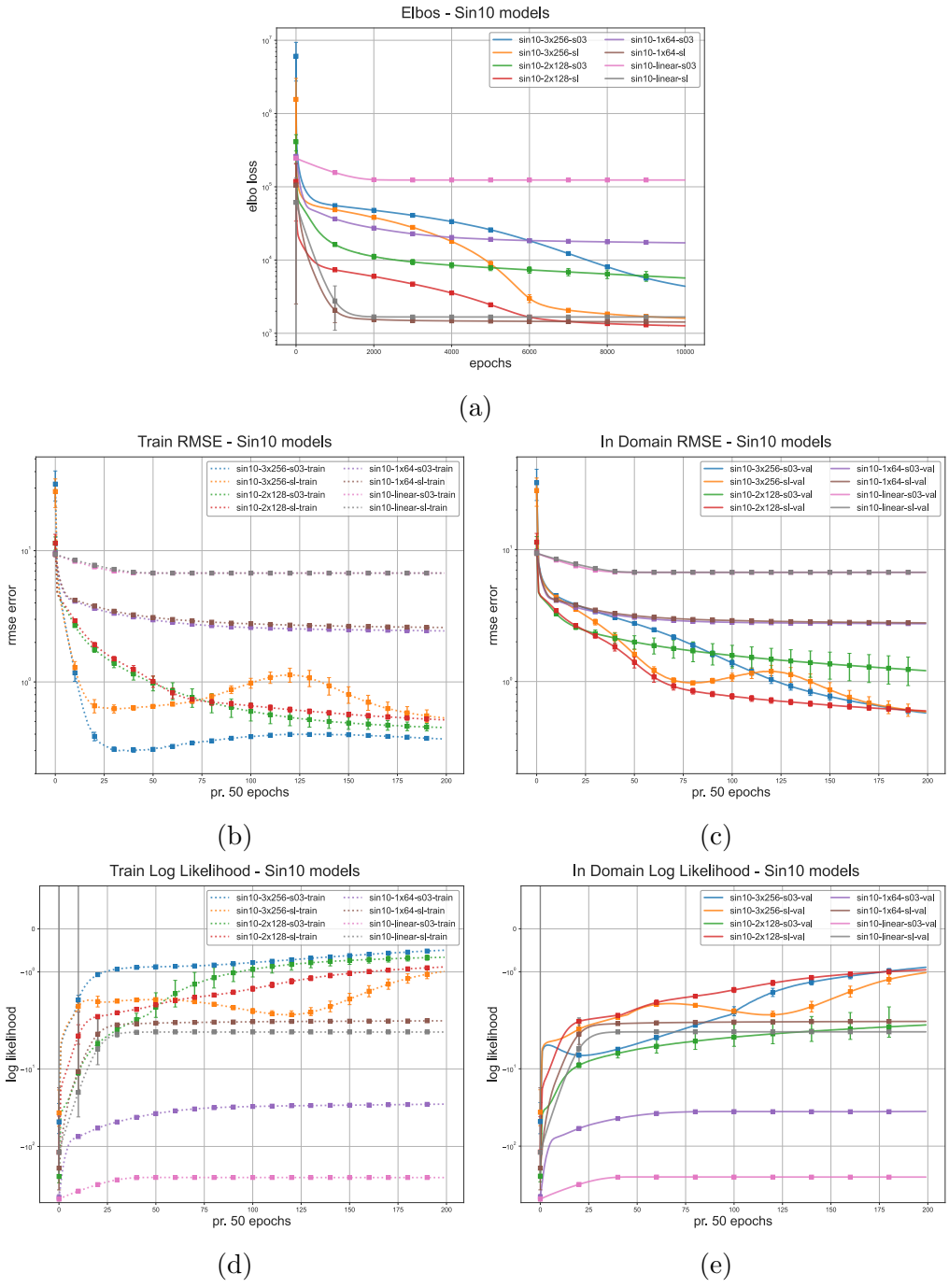


Figure 5.4: Study 2 - Training curves for models trained on the *sin10-s03* dataset. Figure 5.4a shows the elbo loss at each epoch. Figures 5.4b and 5.4c show the RMSE metric for train and validation data, recorded every 50 epochs. Figures 5.4d and 5.4e show the log-likelihood metric for train and validation data, recorded every 50 epochs. The curves are averaged over 10 independent runs, and the error bars show a confidence interval of $2 * SD$ (2 times the standard deviation) to show the differences between random initializations. The y-axis is log-scaled.

(a) Train data

Model	σ	RMSE	MAE	LL
sin10-3x256-s03	0.3±0.0	0.37±0.02	0.29±0.02	-0.5±0.04
sin10-3x256-sl	0.75±0.15	0.53±0.11	0.42±0.08	-0.98±0.19
sin10-2x128-s03	0.3±0.0	0.45±0.05	0.36±0.04	-0.66±0.1
sin10-2x128-sl	0.66±0.07	0.51±0.04	0.41±0.03	-0.88±0.08
sin10-1x64-s03	0.3±0.0	2.46±0.12	1.86±0.07	-28.7±2.71
sin10-1x64-sl	2.79±0.2	2.61±0.09	1.96±0.06	-2.39±0.03
sin10-linear-s03	0.3±0.0	6.68±0.01	5.41±0.01	-247.02±0.45
sin10-linear-sl	6.66±0.42	6.68±0.01	5.41±0.01	-3.32±0.0

(b) In domain data

Model	σ	RMSE	MAE	LL
sin10-3x256-s03	0.3±0.0	0.58±0.06	0.45±0.04	-0.88±0.11
sin10-3x256-sl	0.75±0.15	0.59±0.12	0.47±0.09	-1.02±0.2
sin10-2x128-s03	0.3±0.0	1.21±0.61	0.93±0.46	-2.71±1.96
sin10-2x128-sl	0.66±0.07	0.6±0.05	0.47±0.04	-0.95±0.08
sin10-1x64-s03	0.3±0.0	2.8±0.11	2.09±0.07	-36.88±2.94
sin10-1x64-sl	2.79±0.2	2.85±0.08	2.12±0.05	-2.46±0.03
sin10-linear-s03	0.3±0.0	6.69±0.01	5.46±0.01	-247.76±0.5
sin10-linear-sl	6.66±0.42	6.69±0.01	5.46±0.01	-3.32±0.0

(c) Out of domain data

Model	Likelihood σ	RMSE	MAE	LL
sin10-3x256-s03	0.3±0.0	34.86±3.53	28.65±3.03	-28.01±5.05
sin10-3x256-sl	0.75±0.15	31.45±5.8	26.33±4.51	-6.63±1.63
sin10-2x128-s03	0.3±0.0	25.43±8.82	20.52±7.22	-81.47±25.47
sin10-2x128-sl	0.66±0.07	37.95±3.56	30.97±3.03	-14.03±2.81
sin10-1x64-s03	0.3±0.0	21.03±1.1	17.3±0.92	-1264.77±147.21
sin10-1x64-sl	2.79±0.2	21.26±0.92	17.48±0.78	-17.99±1.63
sin10-linear-s03	0.3±0.0	18.94±0.02	15.39±0.01	-1963.27±3.41
sin10-linear-sl	6.66±0.42	18.84±0.01	15.31±0.01	-6.77±0.01

Table 5.12: Study 2 - Table of results for models trained on the *sin10-s03* dataset. The table shows the Root Mean Squared Error (RMSE), Mean Absolute Error (MAE) and log-likelihood (LL) evaluation metrics for each model. Subtables 5.12a, 5.12b and 5.12c show the results for the train, in-domain test, and out-of-domain test data, respectively. The results are averaged over 10 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) in order to highlight the differences between random initializations.

(a) Train data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
sin10-3x256-s03	0.3±0.0	0.47±0.03	0.32±0.0	1.3±0.14
sin10-3x256-sl	0.75±0.15	0.9±0.17	0.76±0.14	2.04±0.35
sin10-2x128-s03	0.3±0.0	0.53±0.07	0.35±0.03	0.97±0.1
sin10-2x128-sl	0.66±0.07	0.78±0.07	0.67±0.06	1.5±0.2
sin10-1x64-s03	0.3±0.0	0.32±0.0	0.3±0.0	0.35±0.0
sin10-1x64-sl	2.79±0.2	2.96±0.09	2.8±0.09	3.21±0.13
sin10-linear-s03	0.3±0.0	0.3±0.0	0.29±0.0	0.31±0.0
sin10-linear-sl	6.66±0.42	6.67±0.0	6.49±0.05	6.85±0.03

(b) In domain data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
sin10-3x256-s03	0.3±0.0	0.47±0.03	0.32±0.01	1.28±0.19
sin10-3x256-sl	0.75±0.15	0.91±0.18	0.76±0.14	2.03±0.33
sin10-2x128-s03	0.3±0.0	0.53±0.07	0.35±0.03	0.95±0.07
sin10-2x128-sl	0.66±0.07	0.78±0.07	0.67±0.06	1.46±0.15
sin10-1x64-s03	0.3±0.0	0.32±0.0	0.3±0.0	0.35±0.0
sin10-1x64-sl	2.79±0.2	2.96±0.09	2.8±0.08	3.2±0.11
sin10-linear-s03	0.3±0.0	0.3±0.0	0.29±0.0	0.31±0.0
sin10-linear-sl	6.66±0.42	6.67±0.0	6.49±0.03	6.84±0.03

(c) Out of domain data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
sin10-3x256-s03	0.3±0.0	5.01±0.46	2.57±0.27	8.51±1.03
sin10-3x256-sl	0.75±0.15	14.11±2.39	5.22±1.19	35.44±13.8
sin10-2x128-s03	0.3±0.0	2.1±0.67	1.17±0.16	5.86±4.6
sin10-2x128-sl	0.66±0.07	8.13±0.72	3.87±0.48	13.75±1.27
sin10-1x64-s03	0.3±0.0	0.42±0.01	0.36±0.01	0.49±0.02
sin10-1x64-sl	2.79±0.2	3.8±0.12	3.3±0.12	4.43±0.28
sin10-linear-s03	0.3±0.0	0.3±0.0	0.29±0.0	0.31±0.0
sin10-linear-sl	6.66±0.42	6.71±0.0	6.53±0.03	6.89±0.03

Table 5.13: Study 2 - Table of summarized predictive uncertainty for models trained on the *sin10-s03* dataset. Tables 5.13a, 5.13b and 5.13c show the results for the train, in-domain test, and out-of-domain test data, respectively. The tables show the mean uncertainty of all predictive distributions (Mean Predictive SD) for the given dataset, as well as the minimum uncertainty (Min Predictive SD) and maximum uncertainty (Max Predictive SD) of all predictive distributions for the given dataset. The results are averaged over 10 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) in order to highlight the differences between random initializations.

(a) Model weight data

Model	Mean Weight SD	Min Weight SD	Max Weight SD
sin10-3x256-s03	0.87±0.02	0.0±0.0	1.02±0.02
sin10-3x256-sl	0.96±0.0	0.0±0.0	1.01±0.01
sin10-2x128-s03	0.21±0.17	0.0±0.0	1.0±0.01
sin10-2x128-sl	0.86±0.01	0.0±0.0	1.0±0.0
sin10-1x64-s03	0.0±0.0	0.0±0.0	0.01±0.0
sin10-1x64-sl	0.17±0.07	0.01±0.0	0.96±0.0
sin10-linear-s03	0.0±0.0	0.0±0.0	0.0±0.0
sin10-linear-sl	0.08±0.0	0.08±0.0	0.09±0.0

Table 5.14: Study 2 - Table of summarized weight uncertainty for models trained on the *sin10-s03* dataset. The table shows the mean uncertainty of all weight distributions in the models (Mean Weight SD), as well as the minimum uncertainty (Min Weight SD) and maximum uncertainty (Max Weight SD) of all weight distributions in the models. The results are averaged over 10 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) in order to highlight the differences between random initializations.

5.2.2 Results for the *multisin10-s05* dataset

This section presents our results for the *multisin10-s05* dataset. We start by presenting the training curves for the models in figure 5.5. We present further evaluation metrics in table 5.15. Our metrics for summarized predictive uncertainty are presented in table 5.16 and our metrics for summarized weight uncertainty are presented in table 5.17.

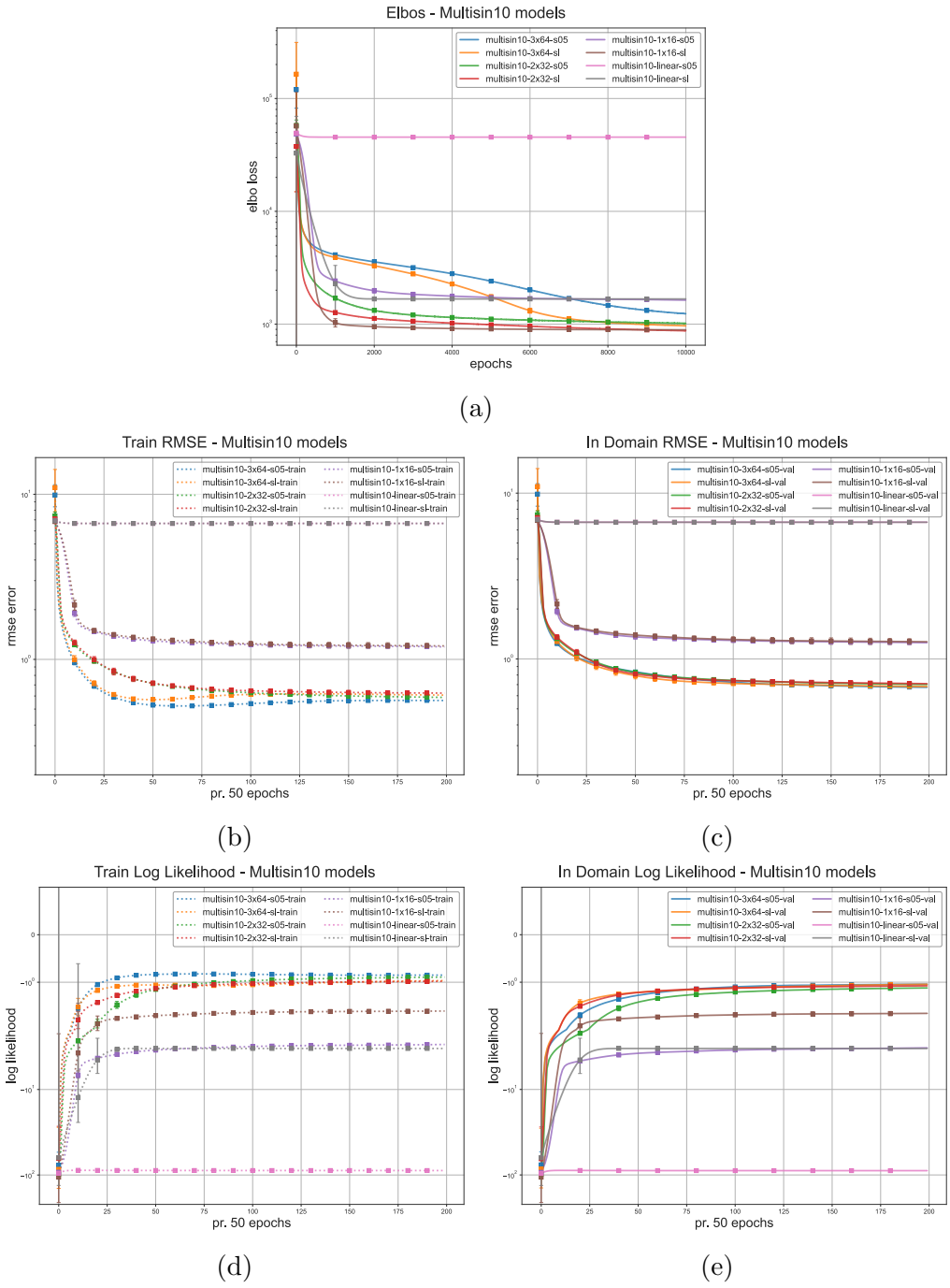


Figure 5.5: Study 2 - Training curves for models trained on the *multisin10-s05* dataset. Figure 5.5a shows the elbo loss at each epoch. Figures 5.5b and 5.5c show the RMSE metric for train and validation data, recorded every 50 epochs. Figures 5.5d and 5.5e show the log-likelihood metric for train and validation data, recorded every 50 epochs. The curves are averaged over 10 independent runs, and the error bars show a confidence interval of $2*SD$ (2 times the standard deviation) to show the differences between random initializations. The y-axis is log-scaled.

(a) Train data

Model	Likelihood σ	RMSE	MAE	LL
multisin10-3x64-s05	0.5±0.0	0.56±0.01	0.45±0.01	-0.85±0.02
multisin10-3x64-sl	0.69±0.05	0.61±0.02	0.48±0.02	-0.97±0.03
multisin10-2x32-s05	0.5±0.0	0.59±0.01	0.47±0.01	-0.9±0.03
multisin10-2x32-sl	0.69±0.05	0.62±0.03	0.5±0.03	-0.98±0.05
multisin10-1x16-s05	0.5±0.0	1.19±0.04	0.94±0.03	-2.96±0.18
multisin10-1x16-sl	1.23±0.11	1.2±0.08	0.95±0.06	-1.6±0.06
multisin10-linear-s05	0.5±0.0	6.73±0.0	5.29±0.0	-90.67±0.13
multisin10-linear-sl	6.71±0.42	6.73±0.0	5.29±0.0	-3.33±0.0

(b) In domain data

Model	Likelihood σ	RMSE	MAE	LL
multisin10-3x64-s05	0.5±0.0	0.68±0.02	0.54±0.02	-1.05±0.04
multisin10-3x64-sl	0.69±0.05	0.69±0.02	0.54±0.01	-1.05±0.02
multisin10-2x32-s05	0.5±0.0	0.7±0.01	0.56±0.01	-1.13±0.03
multisin10-2x32-sl	0.69±0.05	0.71±0.03	0.56±0.02	-1.08±0.04
multisin10-1x16-s05	0.5±0.0	1.25±0.04	0.99±0.03	-3.23±0.21
multisin10-1x16-sl	1.23±0.11	1.26±0.08	1.0±0.06	-1.65±0.07
multisin10-linear-s05	0.5±0.0	6.8±0.01	5.29±0.01	-92.44±0.21
multisin10-linear-sl	6.71±0.42	6.8±0.01	5.29±0.01	-3.34±0.0

(c) Out of domain data

Model	Likelihood σ	RMSE	MAE	LL
multisin10-3x64-s05	0.5±0.0	11.96±0.41	9.56±0.33	-33.61±5.05
multisin10-3x64-sl	0.69±0.05	11.66±0.29	9.32±0.25	-17.3±6.98
multisin10-2x32-s05	0.5±0.0	11.71±0.34	9.36±0.3	-75.21±8.53
multisin10-2x32-sl	0.69±0.05	11.9±0.3	9.54±0.25	-44.7±5.98
multisin10-1x16-s05	0.5±0.0	9.6±0.1	7.67±0.09	-146.72±3.34
multisin10-1x16-sl	1.23±0.11	9.59±0.24	7.67±0.19	-25.42±2.23
multisin10-linear-s05	0.5±0.0	19.41±0.02	15.67±0.02	-743.57±1.57
multisin10-linear-sl	6.71±0.42	19.41±0.02	15.67±0.02	-6.95±0.01

Table 5.15: Study 2 - Table of results for models trained on the *multisin10-s05* dataset. The table shows the Root Mean Squared Error (RMSE), Mean Absolute Error (MAE) and log-likelihood (LL) evaluation metrics for each model. Subtables 5.15a, 5.15b and 5.15c show the results for the train, in-domain test, and out-of-domain test data, respectively. The results are averaged over 10 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) in order to highlight the differences between random initializations.

(a) Train data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
multisin10-3x64-s05	0.5±0.0	0.58±0.01	0.5±0.0	0.86±0.08
multisin10-3x64-sl	0.69±0.05	0.76±0.02	0.69±0.01	1.08±0.1
multisin10-2x32-s05	0.5±0.0	0.56±0.0	0.5±0.0	0.7±0.04
multisin10-2x32-sl	0.69±0.05	0.76±0.03	0.69±0.03	0.91±0.04
multisin10-1x16-s05	0.5±0.0	0.51±0.0	0.49±0.0	0.53±0.0
multisin10-1x16-sl	1.23±0.11	1.25±0.08	1.21±0.08	1.3±0.08
multisin10-linear-s05	0.5±0.0	0.5±0.0	0.49±0.0	0.51±0.0
multisin10-linear-sl	6.71±0.42	6.72±0.0	6.54±0.05	6.91±0.03

(b) In domain data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
multisin10-3x64-s05	0.5±0.0	0.59±0.01	0.5±0.0	0.9±0.08
multisin10-3x64-sl	0.69±0.05	0.76±0.02	0.69±0.01	1.14±0.1
multisin10-2x32-s05	0.5±0.0	0.56±0.0	0.5±0.0	0.7±0.03
multisin10-2x32-sl	0.69±0.05	0.76±0.03	0.69±0.03	0.92±0.05
multisin10-1x16-s05	0.5±0.0	0.51±0.0	0.49±0.0	0.53±0.0
multisin10-1x16-sl	1.23±0.11	1.25±0.08	1.21±0.08	1.3±0.09
multisin10-linear-s05	0.5±0.0	0.5±0.0	0.49±0.0	0.51±0.0
multisin10-linear-sl	6.71±0.42	6.72±0.0	6.55±0.03	6.9±0.03

(c) Out of domain data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
multisin10-3x64-s05	0.5±0.0	1.46±0.13	0.86±0.11	2.45±0.46
multisin10-3x64-sl	0.69±0.05	2.06±0.5	1.24±0.18	4.42±2.48
multisin10-2x32-s05	0.5±0.0	0.96±0.05	0.67±0.05	1.46±0.3
multisin10-2x32-sl	0.69±0.05	1.27±0.07	0.91±0.05	1.83±0.15
multisin10-1x16-s05	0.5±0.0	0.56±0.0	0.52±0.01	0.61±0.0
multisin10-1x16-sl	1.23±0.11	1.38±0.09	1.28±0.09	1.51±0.09
multisin10-linear-s05	0.5±0.0	0.5±0.0	0.49±0.0	0.52±0.0
multisin10-linear-sl	6.71±0.42	6.76±0.0	6.58±0.03	6.94±0.03

Table 5.16: Study 2 - Table of summarized predictive uncertainty for models trained on the *multisin10-s05* dataset. Tables 5.16a, 5.16b and 5.16c show the results for the train, in-domain test, and out-of-domain test data, respectively. The tables show the mean uncertainty of all predictive distributions (Mean Predictive SD) for the given dataset, as well as the minimum uncertainty (Min Predictive SD) and maximum uncertainty (Max Predictive SD) of all predictive distributions for the given dataset. The results are averaged over 10 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) to highlight the differences between random initializations.

(a) Model weight data

Model	Mean Weight SD	Min Weight SD	Max Weight SD
multisin10-3x64-s05	0.69±0.04	0.0±0.0	1.0±0.0
multisin10-3x64-sl	0.81±0.02	0.0±0.0	1.0±0.0
multisin10-2x32-s05	0.08±0.04	0.0±0.0	1.0±0.01
multisin10-2x32-sl	0.28±0.1	0.0±0.0	1.0±0.0
multisin10-1x16-s05	0.01±0.0	0.0±0.0	0.01±0.01
multisin10-1x16-sl	0.02±0.0	0.01±0.0	0.04±0.01
multisin10-linear-s05	0.0±0.0	0.0±0.0	0.01±0.0
multisin10-linear-sl	0.06±0.0	0.05±0.0	0.09±0.0

Table 5.17: Study 2 - Table of summarized weight uncertainty for models trained on the *multisin10-s05* dataset. The table shows the mean uncertainty of all weight distributions in the models (Mean Weight SD), as well as the minimum uncertainty (Min Weight SD) and maximum uncertainty (Max Weight SD) of all weight distributions in the models. The results are averaged over 10 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) to highlight the differences between random initializations.

5.2.3 Results for the *multisin20-s05* dataset

This section presents our results for the *multisin20-s05* dataset. We start by presenting the training curves for the models in figure 5.6. We present further evaluation metrics in table 5.18. Our metrics for summarized predictive uncertainty are presented in table 5.19 and our metrics for summarized weight uncertainty are presented in table 5.20.

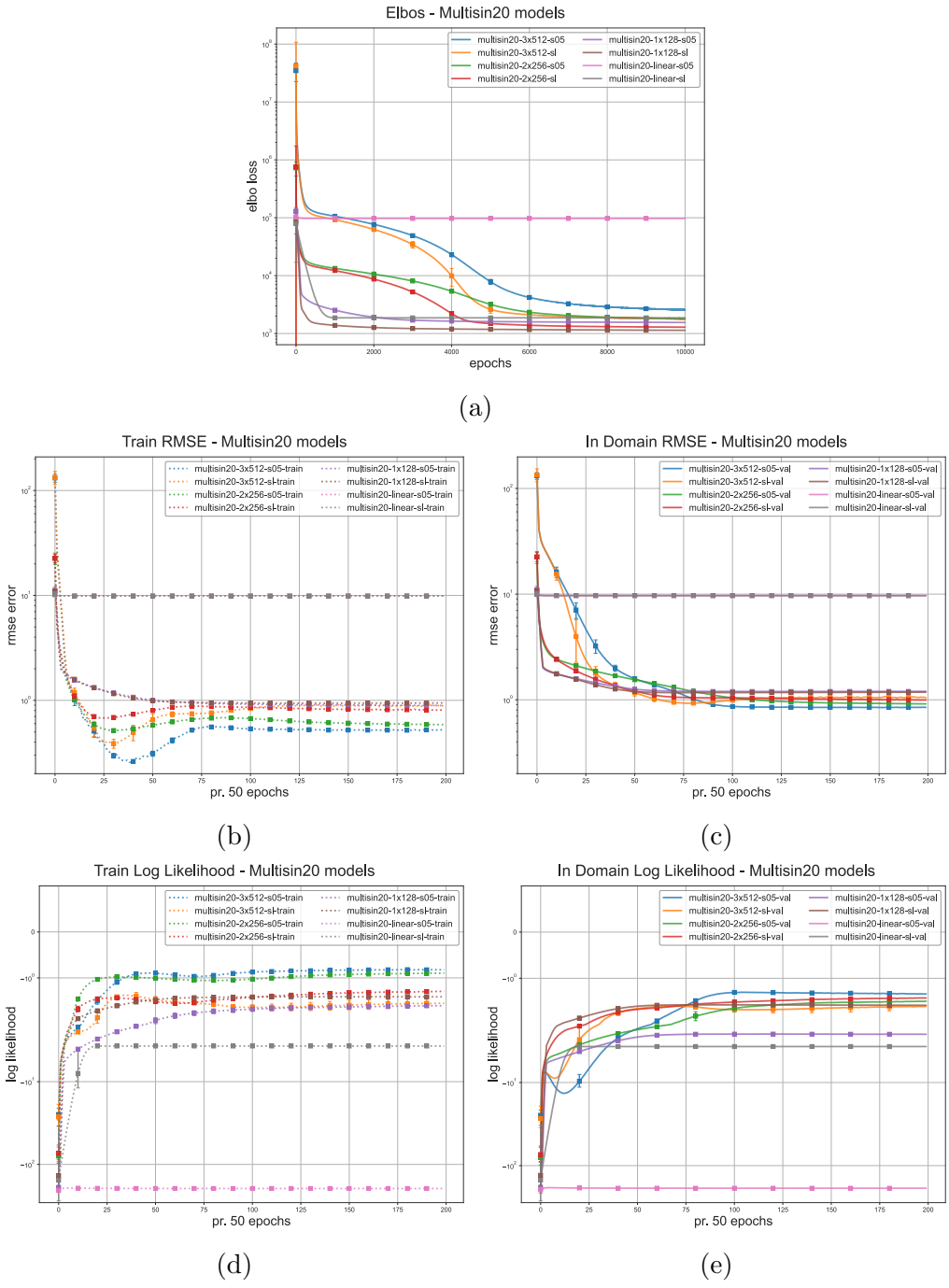


Figure 5.6: Study 2 - Training curves for models trained on the *multisin20-s05* dataset. Figure 5.6a shows the elbo loss at each epoch. Figures 5.6b and 5.6c show the RMSE metric for train and validation data, recorded every 50 epochs. Figures 5.6d and 5.6e show the Log Likelihood metric for train and validation data, recorded every 50 epochs. The curves are averaged over 5 independent runs, and the error bars show a confidence interval of $2*SD$ (2 times the standard deviation) to show the differences between random initializations. The y-axis is log-scaled.

(a) Train data

Model	Likelihood σ	RMSE	MAE	LL
multisin20-3x512-s05	0.5±0.0	0.52±0.01	0.41±0.01	-0.82±0.02
multisin20-3x512-sl	1.33±0.25	0.87±0.11	0.69±0.08	-1.56±0.18
multisin20-2x256-s05	0.5±0.0	0.58±0.01	0.47±0.01	-0.89±0.01
multisin20-2x256-sl	0.98±0.06	0.8±0.02	0.64±0.01	-1.29±0.01
multisin20-1x128-s05	0.5±0.0	0.89±0.01	0.71±0.01	-1.6±0.01
multisin20-1x128-sl	1.08±0.07	0.94±0.01	0.74±0.01	-1.41±0.01
multisin20-linear-s05	0.5±0.0	9.85±0.01	7.8±0.01	-193.94±0.58
multisin20-linear-sl	9.79±0.61	9.85±0.01	7.8±0.01	-3.71±0.0

(b) In domain data

Model	Likelihood σ	RMSE	MAE	LL
multisin20-3x512-s05	0.5±0.0	0.85±0.02	0.68±0.01	-1.33±0.03
multisin20-3x512-sl	1.33±0.25	1.08±0.13	0.85±0.1	-1.63±0.17
multisin20-2x256-s05	0.5±0.0	0.92±0.01	0.74±0.01	-1.51±0.03
multisin20-2x256-sl	0.98±0.06	1.01±0.03	0.8±0.02	-1.44±0.02
multisin20-1x128-s05	0.5±0.0	1.22±0.02	0.96±0.01	-2.68±0.06
multisin20-1x128-sl	1.08±0.07	1.2±0.01	0.95±0.01	-1.6±0.01
multisin20-linear-s05	0.5±0.0	9.96±0.01	7.86±0.01	-198.11±0.31
multisin20-linear-sl	9.79±0.61	9.96±0.01	7.86±0.01	-3.72±0.0

(c) Out of domain data

Model	Likelihood σ	RMSE	MAE	LL
multisin20-3x512-s05	0.5±0.0	15.6±2.6	12.45±2.04	-4.97±0.4
multisin20-3x512-sl	1.33±0.25	15.19±2.49	12.09±1.88	-5.56±2.99
multisin20-2x256-s05	0.5±0.0	16.02±0.28	12.76±0.26	-55.42±8.37
multisin20-2x256-sl	0.98±0.06	15.16±0.25	12.07±0.21	-31.94±1.55
multisin20-1x128-s05	0.5±0.0	13.37±0.07	10.7±0.07	-127.69±1.19
multisin20-1x128-sl	1.08±0.07	13.35±0.07	10.67±0.08	-29.59±0.45
multisin20-linear-s05	0.5±0.0	28.35±0.02	22.55±0.02	-1585.62±2.8
multisin20-linear-sl	9.79±0.61	28.35±0.02	22.55±0.02	-7.34±0.01

Table 5.18: Study 2 - Table of results for models trained on the *multisin20-s05* dataset. The table shows the Root Mean Squared Error (RMSE), Mean Absolute Error (MAE) and log-likelihood (LL) evaluation metrics for each model. Subtables 5.18a, 5.18b and 5.18c show the results for the train, in-domain test, and out-of-domain test data, respectively. The results are averaged over 10 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) in order to highlight the differences between random initializations.

(a) Train data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
multisin20-3x512-s05	0.5±0.0	0.67±0.01	0.52±0.01	1.7±0.32
multisin20-3x512-sl	1.33±0.25	1.67±0.33	1.39±0.28	4.68±2.8
multisin20-2x256-s05	0.5±0.0	0.63±0.01	0.52±0.01	0.85±0.04
multisin20-2x256-sl	0.98±0.06	1.12±0.01	1.01±0.01	1.36±0.19
multisin20-1x128-s05	0.5±0.0	0.56±0.0	0.52±0.0	0.62±0.01
multisin20-1x128-sl	1.08±0.07	1.21±0.01	1.11±0.01	1.33±0.01
multisin20-linear-s05	0.5±0.0	0.5±0.0	0.49±0.0	0.52±0.0
multisin20-linear-sl	9.79±0.61	9.81±0.01	9.53±0.03	10.08±0.04

(b) In domain data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
multisin20-3x512-s05	0.5±0.0	0.66±0.01	0.53±0.01	1.4±0.23
multisin20-3x512-sl	1.33±0.25	1.67±0.3	1.4±0.29	3.52±1.28
multisin20-2x256-s05	0.5±0.0	0.63±0.01	0.52±0.01	0.87±0.13
multisin20-2x256-sl	0.98±0.06	1.12±0.01	1.01±0.01	1.28±0.03
multisin20-1x128-s05	0.5±0.0	0.56±0.0	0.52±0.0	0.62±0.01
multisin20-1x128-sl	1.08±0.07	1.21±0.01	1.12±0.02	1.31±0.01
multisin20-linear-s05	0.5±0.0	0.5±0.0	0.49±0.0	0.51±0.0
multisin20-linear-sl	9.79±0.61	9.81±0.01	9.54±0.07	10.07±0.05

(c) Out of domain data

Model	Likelihood σ	Mean Predictive SD	Min Predictive SD	Max Predictive SD
multisin20-3x512-s05	0.5±0.0	57.93±23.13	19.59±8.45	150.58±54.83
multisin20-3x512-sl	1.33±0.25	32.91±94.77	10.77±27.2	121.73±320.92
multisin20-2x256-s05	0.5±0.0	1.54±0.15	1.17±0.08	2.3±0.33
multisin20-2x256-sl	0.98±0.06	1.93±0.02	1.65±0.04	4.43±1.18
multisin20-1x128-s05	0.5±0.0	0.84±0.0	0.74±0.01	0.95±0.02
multisin20-1x128-sl	1.08±0.07	1.77±0.02	1.56±0.02	2.02±0.02
multisin20-linear-s05	0.5±0.0	0.5±0.0	0.49±0.0	0.52±0.0
multisin20-linear-sl	9.79±0.61	9.87±0.02	9.62±0.04	10.12±0.02

Table 5.19: Study 2 - Table of summarized predictive uncertainty for models trained on the *multisin20-s05* dataset. Tables 5.19a, 5.19b and 5.19c show the results for the train, in-domain test, and out-of-domain test data, respectively. The tables show the mean uncertainty of all predictive distributions (Mean Predictive SD) for the given dataset, as well as the minimum uncertainty (Min Predictive SD) and maximum uncertainty (Max Predictive SD) of all predictive distributions for the given dataset. The results are averaged over 10 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) to highlight the differences between random initializations.

(a) Model weight data

Model	Mean Weight SD	Min Weight SD	Max Weight SD
multisin20-3x512-s05	0.97±0.0	0.0±0.0	1.01±0.0
multisin20-3x512-sl	0.98±0.0	0.0±0.0	1.0±0.01
multisin20-2x256-s05	0.9±0.01	0.0±0.0	1.0±0.0
multisin20-2x256-sl	0.95±0.0	0.0±0.0	1.0±0.0
multisin20-1x128-s05	0.01±0.0	0.0±0.0	0.05±0.01
multisin20-1x128-sl	0.13±0.03	0.0±0.0	0.99±0.0
multisin20-linear-s05	0.0±0.0	0.0±0.0	0.0±0.0
multisin20-linear-sl	0.06±0.0	0.06±0.0	0.1±0.0

Table 5.20: Study 2 - Table of summarized weight uncertainty for models trained on the *multisin20-s05* dataset. The table shows the mean uncertainty of all weight distributions in the models (Mean Weight SD), as well as the minimum uncertainty (Min Weight SD) and maximum uncertainty (Max Weight SD) of all weight distributions in the models. The results are averaged over 10 independent runs, and presented as the mean value $\pm\ 2 * SD$ (2 times standard deviation) to highlight the differences between random initializations.

Chapter 6

Discussion

6.1 Study 1

After observing the training curves in figures 5.1, 5.2, and 5.3, we see that all models seem to converge very consistently to the same elbo value, independently of random initialization. While this generally is a good sign and indicates that the models are not getting stuck in local minima, it could easily result from our prior choice being very restrictive and not allowing the models to explore the parameter space. We can also see that the Elbo curves seem to have little to no noise during optimization. While the reader should note that the y-axis is log normalized, the lack of noise is still surprising. We could assign this to the fact that we take several samples of our Elbo estimate and our large batch size of 512. The lack of noise could also result from our hyperparameter choices being too restrictive and not allowing the model to explore the parameter space. While we have not seen any evidence of this in our experiments, we should be aware of this possibility when discussing our results further.

Another oddity we observe in our training curves is that the model with a high fixed likelihood σ converges to a lower elbo value than the model with the optimal fixed likelihood σ, at least for the larger models (This is not the case for the smaller models used for the *multisin10* dataset). It is counterintuitive that a worse-performing model should converge to a lower elbo value than a better-performing one. This indicates that, at least for the larger models, the prior has a higher influence on the elbo value than the likelihood. If the larger models favor regularization from the prior overfitting the data, then it makes sense that the model with a higher likelihood variance converges to a lower

Elbo value. A solution could be to tweak the KL scaling factor to favor the likelihood more, which could lead to better predictive accuracy. However, Predictive accuracy is not the focus of this study, and we do not believe this oddity will invalidate our results.

Looking at the RMSE curves and the evaluation metrics in tables 5.2, 5.5, and 5.8, we see indications of our hypothesis being correct. Viewing the RMSE and MAE values, we see that the models with a small fixed likelihood σ consistently achieve the lowest error on the train data by a large margin while performing horribly on in-domain test data. This is a clear indication of overfitting. Similarly, the models with a large fixed likelihood σ consistently achieve the highest error on the train data, failing to fit the data properly. We also see very similar performance on the in-domain test data, which clearly indicates underfitting. Examining the weight uncertainty in tables 5.4, 5.7 and 5.10, we can see that the average weight uncertainty (Mean Weight SD) is much lower for the models with a small fixed likelihood σ than for the other models, indicating that too small values of σ also leads to overconfidence in the posterior. Viewing the average weight uncertainty of the models with a large fixed σ, we do not see a large increase in weight uncertainty. This is not exactly unexpected, as underfitting does not directly lead to higher weight uncertainty. Also, since the model is unable to fit the data properly, it has likely only been able to optimize the complexity cost of the Elbo by keeping the posterior distribution as close as possible to the original prior.

Once again, examining the evaluation metrics, we see that the models with a fixed "optimal" likelihood σ score the lowest RMSE and MAE values on both train and in domain test data, as expected. What is interesting, however, is that the models with an inferred likelihood σ perform comparably to the models with a fixed "optimal" likelihood σ. While the error on train data is slightly worse, the RMSE, MAE, and LL values on in-domain test data are almost equivalent. This is a promising result, as it indicates that one can infer the likelihood variance from the data with little to no loss in performance.

As one would expect, all models for all datasets score very high RMSE and MAE values on out-of-domain test data. While poor out-of-domain performance is expected, a Bayesian neural network is also expected to be able to express high uncertainty in such cases. Examining the predictive uncertainty estimates presented in tables 5.3, 5.6, and 5.9, we can see that the models with a small fixed likelihood σ have low average predictive uncertainties (Mean Predictive SD), and are therefore very confident in their predictions, even on out of domain data. We can also see that the models with inferred likelihood σ are able to express higher uncertainty on out-of-domain data than the models with a fixed "optimal" likelihood σ.

We must also examine an oddity we observed in the *sin10* dataset. Looking at the evaluation metrics for models trained on the *sin10* dataset in table 5.2, we see that the models with a small fixed likelihood σ and a high fixed likelihood σ score significantly lower RMSE and MAE values on out of domain data than the two other models. This is highly unexpected. After examining samples of the predictive distributions for the models, we believe that this oddity is a result of a quirk in the data. The *sin10* dataset combines differently scaled sine waves, all centered around 0. Consequently, the y-space of the dataset is limited to a set interval around 0 and is identical for in-domain and out-of-domain data. The underfitted model always seems to output a predictive distribution centered around 0, which covers the entire y-space. Because of this, it often predicts values close to 0 and within the y-space of the out-of-domain data. The overfitted model, on the other hand, has a very narrow predictive distribution. However, it always predicts a value within the y-space of the train data, which is also the y-space of the out-of-domain data. We can see that while the overfitted model has a lower RMSE on the out-of-domain data, it still has a very low log-likelihood, which tells us that the predictive distribution seldom overlaps with the true data distribution. While the out-of-domain RMSE values are strange and unexpected in this case, we believe it to be a quirk of the dataset and not descriptive of the model's ability to generalize to out-of-domain data.

6.2 Study 2

Just like the previous experiments, when examining the Elbo curves in figures 5.4a, 5.5a and 5.6a, we see that the models converge consistently, with little variance between runs. This is a good indication that our models converge to the same solution and that our results are consistent across random initializations. However, as with the previous experiment, we should be wary that the lack of noise in the Elbo curves could result from our models being too restricted to explore the full space of possible solutions.

Examining the evaluation metrics in tables 5.12, 5.15 and 5.18, we can observe that the smaller misspecified models produce predictions with high RMSE and MAE values across all datasets, which is to be expected. Looking closer at the log-likelihood (LL) metrics, we can see that the smaller models with a fixed σ have very low log-likelihood values and high error, indicating that the model produces wrong predictions with high confidence. The smaller models with a learnable σ, however, have much higher log-likelihood values, indicating that while the model produces wrong predictions, it also produces a higher uncertainty in the predictions, leading to more overlap with the true

data distribution and thus a higher log-likelihood. We can see the reason for the higher uncertainty by examining the likelihood σ values. For the smaller models with a learnable σ, the likelihood σ values have scaled to a high value, growing higher as the number of parameters decreases.

Examining the summarized weight uncertainty in tables 5.14, 5.17 and 5.20. We can see that the mean weight uncertainty (Mean Weight SD) falls drastically as the number of parameters decreases. While a reduction in uncertainty is expected as the models grow smaller just from the fact that there are fewer weights to be uncertain about, the maximum weight uncertainty falls to near zero for the smallest models. This is a clear indication that we have created overconfident posterior distributions. Further examining the summarized predictive uncertainty in tables 5.13, 5.16 and 5.19, we can see that the mean predictive uncertainty (Mean Predictive SD), minimum predictive uncertainty (Min Predictive SD) and maximum predictive uncertainty (Max Predictive SD) for the smaller misspecified models vary little between train, in-domain test and out-of-domain test data. We can also see that almost all of the uncertainty in the predictive distributions of the smaller models stem from the σ values in the likelihood, which makes sense considering the low weight uncertainty. This also means that, at least for the very misspecified models, the learnable σ values have aided significantly in expressing a higher uncertainty more in line with the error in their predictions.

On another note, examining once again the evaluation metrics we can see that for the *multisin10* and *multisin20* datasets, the first reduction in model size (from *multisin10-3x64-** to *multisin10-2x32-** and from *multisin20-3x512-** to *multisin20-2x256-**) consistently lead to very little loss in performance. This is unexpected and could point to a shortcoming in our model selection. Since we can obtain almost the same result with a significantly smaller model, we could have saved much computing time had we used these models from the start. It would also have made training with MCMC much more feasible. However, looking closer at the models for the *sin10* dataset in table 5.12, we can see that the results differ slightly. The *sin10-2x128-s03* model with a fixed σ performs noticeably worse than the *sin10-3x256-s03* model on the train data, and much worse on the in-domain data, with a high variance between random initializations. This issue is not present in the *sin10-2x128-sl* model with a learnable σ, which performs almost the same as the *sin10-3x256-s03* and *sin10-3x256-sl* models. This could indicate that the learnable σ helps to stabilize the training of the smaller models, at least with our setup. However, seeing as this result is not present in the other two datasets, it could easily be another quirk of this dataset, which we have seen in previous experiments. Therefore, we are weary of drawing conclusions from this particular result.

Chapter 7

Conclusion

In this thesis, we have explored regression with Bayesian neural networks and the impact of the likelihood function on the performance of the model. More specifically, we have investigated the Gaussian likelihood function commonly used in regression tasks and the impact of the variance parameter σ on the resulting posterior and predictive distributions. We have also investigated the feasibility of setting a prior over the σ parameter and inferring its probability distribution from the data.

Through experiments where we have trained several Bayesian neural network regressors using variational inference on three different synthetic datasets, we have showcased the importance of a well-specified Gaussian likelihood function. Our results indicate that very narrow Gaussian likelihood functions parameterized by a low σ parameter leads to overfitting. Similarly, very wide Gaussian likelihood functions parameterized by a high σ parameter show signs of underfitting.

Additionally, our results show that models that inferred the σ parameter from the data performed on par when compared with models parameterized with an "optimal" fixed σ parameter, set to the true noise level of the data. This indicates that the inferred σ parameter is a good alternative, removing the need to set the σ parameter manually while causing little to no loss in performance.

Furthermore, through additional experiments, we have gained further insight into how the inferred σ parameter affects the posterior distribution and the uncertainty estimates of misspecified Bayesian neural networks. Our results indicate that misspecified Bayesian neural networks, at least when trained using variational inference, struggle to express any meaningful uncertainty in their weight posterior distributions. Therefore, misspecified

models with a fixed σ parameter tend to produce overconfident predictive distributions. However, when the σ parameter is inferred from the data, its value scales to higher values for misspecified models, which in turn helps compensate for their overconfident uncertainty estimates. For Bayesian neural networks, knowing when we do not know is just as important as knowing, and the inferred σ parameter helps us achieve this. This property is indeed exciting and warrants further investigation.

It is important to note that our experiments are far from perfect, and our study has some limitations, which we will discuss in the next section. We want to reiterate that our findings are not conclusive. Bayesian neural networks are intricate and challenging to train and tune, and our experiments are limited in scope and size. Furthermore, creating a stable environment for our study proved difficult, considering the stochastic nature of Bayesian neural networks and their sensitivity to hyperparameters. We have done our best to create a stable environment for our experiments, but we can not guarantee that some unknown factor will not affect our results. In order to draw any meaningful conclusions about our findings, we would need to conduct more experiments and gather more data. Most importantly, we would need to conduct experiments on more complex models and datasets, as well as other types of data functions and prior distributions, to see if our results generalize to other setups than our own. Additionally, we would also need to run the experiments using MCMC sampling. Unlike variational inference, which only creates a simplified approximation of the posterior, MCMC sampling lets us sample directly from the posterior distribution. Therefore, we can assume that MCMC sampling would produce models with more expressive posterior distributions, which could challenge some of our findings. Unfortunately, we were unable to do this due to complications during development and the time and computational resources required to run MCMC sampling.

We can, however, say that our findings do give further insight into the Gaussian likelihood function and its impact on model performance and learning, as well as the feasibility of inferring the σ parameter from the data and the possible benefits of doing so. While our findings are inconclusive, we have laid the groundwork for further research and understanding of this topic.

7.1 Limitations

MCMC Implementation and Missing Experiments

Originally, we intended to run all experiments using Variational Inference and MCMC. However, due to problems with our selected MCMC implementation and time constraints,

we could only run the experiments using Variational Inference. Our issues stemmed from compatibility issues between the Pyro MCMC implementation and our compute server. We also needed more flexibility in the Pyro MCMC implementation, which made it difficult to run the experiments we wanted to run. We could have avoided many of these issues if we had instead used the Numpyro MCMC framework, a more recent MCMC framework built on top of JAX and Numpy. However, this would have required us to rewrite much of the TYXE codebase, which we decided was not worth the effort given the time constraints.

Imperfect metrics for evaluating model uncertainty and assumptions about the posterior distribution from limited data

The evaluation metrics we have used to evaluate the uncertainties of our model posteriors are simple averages of the uncertainty of individual weight parameters. While the metric offers some insight into the uncertainty of the model, it is a limited view of the true model uncertainty. A more thorough analysis of the posterior distribution would be required to draw any meaningful conclusions, but this proves to be a significant challenge due to the high dimensional nature of the Bayesian neural network posterior distribution. Many of the conclusions we have drawn in this thesis are based on a combination of this limited analysis of the posterior distribution and our other metrics primarily based on the predictive distribution.

While we can get some meaningful insight into the posterior by looking at the predictive distribution, it is important to note that we are making assumptions based on a limited picture of the true posterior distribution. Therefore, we underline that our results are inconclusive and that either a more thorough analysis of the posterior distribution or more data is required to draw meaningful conclusions.

Small variances in predictive distribution uncertainty metrics

The metrics we have used to evaluate the uncertainty of the predictive distribution are based on samples drawn from the predictive distribution. We have used 10,000 samples from the predictive distribution to calculate the mean and standard deviation of the predictive distribution. This number is insufficient to produce a perfect estimate, so minor variances occur in the metrics. This is not especially noticeable or problematic when comparing models to each other. It can, however, cause some strange results.

Generally, it causes the uncertainty metrics for the predictive distribution to be slightly lower than expected. Because of this, we have been careful not to draw any conclusions based on these metrics alone. Instead, we use them as an indication of overall predictive uncertainty and as a supplement to our other metrics.

7.2 Future Work

As discussed in the previous section, some parts of our study were left out due to time constraints and other issues.

Firstly, it would be interesting to see the results of our experiments using MCMC sampling instead of variational inference. While stochastic variational inference is a very efficient way of training Bayesian neural networks, it only creates a simplified approximation of the true posterior distribution. It would be interesting to see if our results persist for MCMC inference, or if the observed results are limited to variational inference. The NumPyro framework [55] is a more recent MCMC framework built on top of JAX and Numpy, which boasts a more flexible and stable implementation than Pyro, as well as significant improvements in speed and performance. Making use of the improvements in NumPyro would make running similar experiments with MCMC sampling much more feasible.

It would also be interesting to see similar experiments run on more models and different datasets. Seeing more data and results would help to draw more meaningful conclusions and to see if the results presented in this thesis generalize to other settings. Similarly, experimenting with more complex model priors other than mean-field Gaussian would also be interesting.

Bibliography

[1] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[2] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989. ISSN 1435-568X. doi: 10.1007/BF02551274. URL https://doi.org/10.1007/BF02551274.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[4] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation, 2014.

[5] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[6] Chaochao Lu and Xiaoou Tang. Surpassing human-level face verification performance on lfw with gaussianface, 2014.

[7] Florian Schroff, Dmitry Kalenichenko, and James Philbin. FaceNet: A unified embedding for face recognition and clustering. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, jun 2015. doi: 10.1109/cvpr.2015.7298682. URL https://doi.org/10.1109%2Fcvpr.2015.7298682.

[8] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[11] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, and G. Zweig. The microsoft 2016 conversational speech recognition system. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, mar 2017. doi: 10.1109/icassp.2017.7953159. URL `https://doi.org/10.1109%2Ficassp.2017.7953159`.

[12] Jiwoong Choi, Dayoung Chun, Hyun Kim, and Hyuk-Jae Lee. Gaussian yolov3: An accurate and fast object detector using localization uncertainty for autonomous driving, 2019.

[13] Daniel Kermany, Michael Goldbaum, Wenjia Cai, Carolina Valentim, Hui-Ying Liang, Sally Baxter, Alex McKeown, Ge Yang, Xiaokang Wu, Fangbing Yan, Justin Dong, Made Prasadha, Jacqueline Pei, Magdalena Ting, Jie Zhu, Christina Li, Sierra Hewett, Jason Dong, Ian Ziyar, and Kang Zhang. Identifying medical diagnoses and treatable diseases by image-based deep learning. *Cell*, 172:1122–1131.e9, 02 2018. doi: 10.1016/j.cell.2018.02.010.

[14] Victoria Mar and Peter Soyer. Artificial intelligence for melanoma diagnosis: How can we deliver on the promise? *Annals of oncology : official journal of the European Society for Medical Oncology*, 29, 05 2018. doi: 10.1093/annonc/mdy193.

[15] Dmitrii Bychkov, Nina Linder, Riku Turkki, Stig Nordling, Panu Kovanen, Clare Verrill, Margarita Walliander, Mikael Lundin, Caj Haglund, and Johan Lundin. Deep learning based tissue analysis predicts outcome in colorectal cancer. *Scientific Reports*, 8, 02 2018. doi: 10.1038/s41598-018-21758-3.

[16] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, jan 2014. ISSN 1532-4435.

[17] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks, 2015.

[18] Radford M. Neal. *Monte Carlo Implementation*, pages 55–98. Springer New York, New York, NY, 1996. ISBN 978-1-4612-0745-0. doi: 10.1007/978-1-4612-0745-0_3. URL https://doi.org/10.1007/978-1-4612-0745-0_3.

[19] Ethan Goan and Clinton Fookes. Bayesian neural networks: An introduction and survey. In *Case Studies in Applied Bayesian Data Science*, pages 45–87. Springer International Publishing, 2020. doi: 10.1007/978-3-030-42553-1_3. URL https://doi.org/10.1007%2F978-3-030-42553-1_3.

[20] Matt Hoffman, David M. Blei, Chong Wang, and John Paisley. Stochastic variational inference, 2013.

[21] Vincent Fortuin, Adrià Garriga-Alonso, Sebastian W. Ober, Florian Wenzel, Gunnar Rätsch, Richard E. Turner, Mark van der Wilk, and Laurence Aitchison. Bayesian neural network priors revisited, 2022.

[22] Daniele Silvestro and Tobias Andermann. Prior choice affects ability of bayesian neural networks to identify unknowns, 2020.

[23] Mariia Vladimirova, Jakob Verbeek, Pablo Mesejo, and Julyan Arbel. Understanding priors in Bayesian neural networks at the unit level. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6458–6467. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/vladimirova19a.html.

[24] Laurent Valentin Jospin, Hamid Laga, Farid Boussaid, Wray Buntine, and Mohammed Bennamoun. Hands-on bayesian neural networks—a tutorial for deep learning users. *IEEE Computational Intelligence Magazine*, 17(2):29–48, may 2022. doi: 10.1109/mci.2022.3155327. URL https://doi.org/10.1109%2Fmci.2022.3155327.

[25] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943. ISSN 1522-9602. doi: 10.1007/BF02478259. URL https://doi.org/10.1007/BF02478259.

[26] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[27] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark, 2022.

[28] Abien Fred Agarap. Deep learning using rectified linear units (relu), 2019.

[29] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. doi: 10.1038/323533a0. URL https://doi.org/10.1038/323533a0.

[30] Sebastian Ruder. An overview of gradient descent optimization algorithms. *ArXiv*, abs/1609.04747, 2016. URL https://api.semanticscholar.org/CorpusID:17485266.

[31] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[32] Daniele Silvestro and Tobias Andermann. Prior choice affects ability of bayesian neural networks to identify unknowns, 2020.

[33] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks, 2021.

[34] Pavel Izmailov, Sharad Vikram, Matthew D. Hoffman, and Andrew Gordon Wilson. What are bayesian neural network posteriors really like?, 2021.

[35] Jennifer A. Hoeting, David Madigan, Adrian E. Raftery, and Chris T. Volinsky. Bayesian model averaging: A tutorial. *Statistical Science*, 14(4):382–401, 1999. ISSN 08834237. URL http://www.jstor.org/stable/2676803.

[36] Martin Magris and Alexandros Iosifidis. Bayesian learning for neural networks: an algorithmic survey, 2022. URL https://arxiv.org/abs/2211.11865.

[37] Charles J. Geyer. Practical markov chain monte carlo. *Statistical Science*, 7(4):473–483, 1992. ISSN 08834237. URL http://www.jstor.org/stable/2246094.

[38] Charles J. Geyer. *Introduction to Markov Chain Monte Carlo*, pages 3–48. CRC Press, May 2011. ISBN 9781420079418. doi: 10.1201/b10905-2.

[39] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953. doi: 10.1063/1.1699114. URL http://link.aip.org/link/?JCP/21/1087/1.

[40] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970. ISSN 00063444. URL http://www.jstor.org/stable/2334940.

[41] Radford Neal. Mcmc using hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 06 2012. doi: 10.1201/b10905-6.

[42] Matthew D. Hoffman and Andrew Gelman. The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo, 2011. URL https://arxiv.org/abs/1111.4246.

[43] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. An introduction to variational methods for graphical models. *Machine Learning*, 37(2):183–233, Nov 1999. ISSN 1573-0565. doi: 10.1023/A:1007665907178. URL https://doi.org/10.1023/A:1007665907178.

[44] S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79 – 86, 1951. doi: 10.1214/aoms/1177729694. URL https://doi.org/10.1214/aoms/1177729694.

[45] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518): 859–877, apr 2017. doi: 10.1080/01621459.2017.1285773. URL https://doi.org/10.1080%2F01621459.2017.1285773.

[46] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2014.

[47] Cheng Zhang, Judith Butepage, Hedvig Kjellstrom, and Stephan Mandt. Advances in variational inference, 2018.

[48] Alex Graves. Practical variational inference for neural networks. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011. URL https://proceedings.neurips.cc/paper_files/paper/2011/file/7eb3c8be3d411e8ebfab08eba5f49632-Paper.pdf.

[49] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models, 2014.

[50] Diederik P. Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick, 2015.

[51] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. *SVI Part IV: Tips and Tricks*, 2021. Available: https://pyro.ai/examples/svi_part_iv.html [Accessed: July 8th, 2023].

[52] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL https://arxiv.org/abs/1912.01703.

[53] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming, 2018. URL https://arxiv.org/abs/1810.09538.

[54] Hippolyt Ritter and Theofanis Karaletsos. Tyxe: Pyro-based bayesian neural nets for pytorch, 2021. URL https://arxiv.org/abs/2110.00276.

[55] Du Phan, Neeraj Pradhan, and Martin Jankowiak. Composable effects for flexible and accelerated probabilistic programming in numpyro, 2019.

Appendix A

Modifications to the TyXe Library

As mentioned in section 4.6, we had to modify the TyXe library in order to use learnable variance parameters in the Gaussian likelihood function. While being able to set a prior distribution on the variance parameter is an intended feature of the library, seeing as it was briefly mentioned in the original paper [54], we found that this particular feature was ignored at prediction time, causing the likelihood function to draw samples from the prior distribution instead of the posterior. Because of this, we had to make some modifications to the library in order to use it for our purposes.

The most significant change we made was to the `VariationalBNN` class in the TyXe library. The original implementation gathered predictions from only the neural network, and then, using the sigma value from the likelihood, they calculated the mean and standard deviation of the predictive distribution directly. This did not pose a problem for a fixed sigma value, but when we introduced a learnable sigma parameter with a variance of its own, the calculations were no longer correct. We have modified the code to gather predictions from the predictive distribution directly by passing predictions through the likelihood function. This also lets us pass the guide trace to the likelihood function, which is necessary for the learnable sigma parameter to work. After gathering predictions, we calculate the mean and standard deviation of the predictive distribution and return these values. However, this approach requires a large number of samples from each predictive distribution to get a reasonable estimate of the mean and standard deviation.

```
class VariationalBNN(_SupervisedBNN):
...

    def guided_forward(self, *args, guide_tr=None, **kwargs):
        if guide_tr is None:
            guide_tr = poutine.trace(self.guide).get_trace(*args, **kwargs)

        pred = poutine.replay(self.net, trace=guide_tr)(*args, **kwargs)
        pred = poutine.replay(self.likelihood, trace=guide_tr)(pred)

        return pred

    def predict(self, *input_data, num_predictions=1000, guide_traces=None):
        if guide_traces is None:
            guide_traces = [None] * num_predictions

        preds = []
        with torch.autograd.no_grad():
            for trace in guide_traces:
                pred = self.guided_forward(*input_data, guide_tr=trace)
                preds.append(pred)

        predictions = torch.stack(preds)
        mean = predictions.mean(dim=0)
        std = predictions.std(dim=0)

        return mean, std

...
```